Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Tesis de Licenciatura

Diseño e implementación de técnicas de planificación de tiempo real en un sistema operativo de tiempo compartido

Integrantes:

Viviana Albarracín – L.U. 727/87

Maria Eugenia Nazar – L.U. 405/84

Director:

Dr. Gabriel A. Wainer

1. RESUMEN

En el presente trabajo, tomando un sistema operativo multitarea (Minix) convertido en un sistema de tiempo real, se agregan nuevas funcionalidades de polimorfismo de desempeño a las planificaciones clásicas de procesador. A la vez, se modifica el sistema operativo para que soporte nuevas planificaciones. Posteriormente, se compara el comportamiento de dicho sistema una vez sufridas estas modificaciones. Y para ello, se construye una herramienta de benchmark basada en el Hartstone, en la cual, dado un archivo de entrada, se ejecutan diversos experimentos arrojando los resultados en un archivo de salida para poder ser manipulados y analizados.

2. ABSTRACT

In the present paper, using a multitasking operating system (Minix) already transformed in a real – time operating system, new functions are added to the traditional scheduling of the processor. Likewise, the operating system is modified so that it can support new scheduling. Later on, once it is modified, the behavior of such operating system is analyzed under different scheduling by means of a benchmark tool based on the Hartstone, where, given an input file various experiments are executed with the results being yielded in outputs files.

3. INTRODUCCIÓN

En este trabajo se analiza el comportamiento de un sistema operativo de tiempo real frente a distintas planificaciones del procesador. De los distintos módulos que componen un sistema operativo se focaliza el estudio en la administración del procesador. Se utilizan las planificaciones para comparar la perfomance de las mismas según distintas cargas de trabajo. Esto se realiza a través de una herramienta que permite lanzar lotes de prueba y recorrer las planificaciones arrojando el resultado en un archivo de salida. Los datos de entrada son incorporados desde un archivo en donde el usuario puede ingresar cualquier conjunto de tareas: periódicas y aperiódicas, indicando el tiempo de ejecución y periodo para cada una y cualquier tipo de combinación de las mismas.

Los algoritmos a comparar son los de mayor difusión entre la bibliografía del tema. A saber :

- Tasa Monotónica,
- Deadline Driven (DD)
- Least Laxity First (LLF).

Además se compara cómo funcionan estas soluciones clásicas con soluciones como polimorfismo de desempeño, cálculos imprecisos y algoritmos mixtos.

Para lograr los objetivos anteriormente propuestos se eligió un sistema operativo de Tiempo Compartido (Minix, un clon de Unix ™) con servicios extendidos para proveer facilidades de tiempo real.

A este sistema operativo se le implementaron llamadas al sistema y modificaciones al algoritmo de planificación que permitieron cumplir con los objetivos del trabajo.

La segunda etapa del presente trabajo es la construcción de una herramienta para testear el comportamiento de los algoritmos de planificación implementados en la primera etapa. La herramienta debe ser capaz de interpretar un archivo de entrada con un conjunto de tareas a

ejecutarlas de acuerdo a sus características y recolectar datos estadísticos para grabarlos en un archivo.

Cada prueba que realice la herramienta consta de ejecutar el conjunto de tareas por un tiempo determinado en cada planificación. Antes de cambiar de algoritmo de planificación, recolectará las estadísticas pertinentes.

Las características del conjunto de datos no depende de la herramienta, sino de los datos de entrada, con lo que queda bajo la responsabilidad del programador planear el conjunto de tareas a testear y con esto se logra que se puedan testear los algoritmos bajo distintos punto de vista (comportamiento con tareas periódicas cortas, largas, con aperiódicas, etc.)

La herramienta corre varias veces el mismo conjunto de datos con el mismo algoritmo, cada vez que comienza un ciclo se proveen tres testeos de estrés: uno para ver si el sistema es capaz de soportar incrementos cortos de tiempo y cambios de contextos rápidos; otro para testear la capacidad del sistema para manejar un carga incremental pero balanceada y un tercero para testear la habilidad del sistema para manejar grupos de tareas grandes.

Otra posibilidad que brinda la herramienta propuesta para este trabajo, es que el usuario define la tarea periódica y aperiódica que se ejecuta. Con lo que se logra gran versatilidad en cuanto a las pruebas que se pueden realizar.

4. CONCEPTOS

4.1. Sistemas de Tiempo Real

Los Sistemas de Tiempo Real son aquellos que controlan procesos en el mundo real, como los sistemas para el control de tráfico aéreo, centrales de energía nuclear, robots, control de condiciones ambientales en el interior del edificio, cadenas de producción en fabricas, vuelos de vehículos espaciales, etc.

Como su flujo de datos depende de eventos que ocurren en el mundo real, la correctitud de los resultados obtenidos es tan importante como el momento en que éstos se obtienen. Los sistemas operativos multitarea no proporcionan un soporte adecuado para los sistemas con estas características. Generalmente esos sistemas soportan programación concurrente, sincronización de tareas y comunicaciones entre tareas. Permiten compartir recursos y servicios, pero no incorporan facilidades para restricciones de tiempo [WAI93]. Así como la programación secuencial no es adecuada al tipo de problemas de programación concurrente por ser aplicaciones paralelas, las soluciones tradicionales de programación concurrentes no se adaptan al desarrollo de aplicaciones de tiempo real ya que las estructuras utilizadas se basan en la noción de eventualidad de ocurrencias de eventos, sin considerar el momento en el que estos ocurren [WAI97].

Existe una gran variedad de sistemas usados para controlar procesos llamados de tiempo real. A pesar de eso siguen existiendo confusiones acera del término "tiempo real" ya que muchas veces estos sistemas se desarrollan usando técnicas similares a las usadas en sistemas tradicionales [WAI97]. Una característica importante de los sistemas de tiempo real está relacionada con el tiempo de procesamiento de las aplicaciones. Generalmente en los sistemas tradicionales no es crítico un control exacto del tiempo. Se puede querer ejecutar a mayor velocidad, pero si se retrasan los resultados, igual siguen siendo válidos. Si consideramos un sistema que controla un avión a reacción, este sistema deberá tener resultados a tiempo para que se puedan tomar decisiones correctas. Si los resultados se atrasan puede ser que ya no sirvan. Por este motivo se dice que los sistemas deben actuar en tiempo real.

En los sistemas tradicionales se trata de optimizar diversos factores, de acuerdo a las características de la aplicación. Se puede tratar de mejorar la utilización global de los recursos de la computadora, los tiempos de espera de los usuarios, el aprovechamiento de los recursos más caros del sistema, o la atención a usuarios prioritarios. Para lograr esto, el orden de ejecución del las tareas del sistema esta determinado por factores internos. En cambio en un sistema de tiempo real se deben atender las distintas demandas a medida que se presentan debiendo considerar los eventos externos, lo cual impide la organización del sistema en base a factores internos. Esta es otra característica importante de los sistemas de tiempo real. En resumen, en los sistemas de tiempo real se deben obtener respuestas en un tiempo máximo determinado, conocido como la "meta" de la tarea y el orden de ejecución de las tareas depende de la ocurrencia de eventos que suceden en el mundo real. Otro factor a tener en cuenta es la confiabilidad ya que un error en estos tipos de sistemas podría resultar en grandes pérdidas de materiales e incluso de vidas según el tipo de aplicaciones que controlen.

Otro enfoque define a los sistemas de tiempo real como los sistemas en los cuales la ejecución de las tareas está determinada por el paso del tiempo u ocurrencia de eventos externos, y los resultados obtenidos pueden depender del momento en que se ejecutaron, o del tiempo en que se tardó en obtenerlos.

Se debe poner especial atención a la expresión "ocurrencia de eventos externos" de la definición, ya que en algún sentido todos los programas de alguna manera responden ante estos eventos. Por ejemplo en el caso de sistemas interactivos, existe un usuario escribiendo en una terminal, hasta podría decirse que los sistemas batch son manejados por la ocurrencia de eventos, ya que una ejecución comienza por haberse cumplido determinada hora. Para acotar el alcance de la definición tomaremos como "evento externo" a un estímulo aplicado al sistema desde el mundo real, o que es una respuesta externamente observable que el sistema realiza sobre el ambiente físico que lo rodea.

> Es fácil comprobar que el sistema de control de una aeronave es un sistema de tiempo real según la definición: ejecuta rutinas de control cada determinado tiempo y ante la ocurrencia de eventos externos (cambios de posición de la aeronave debido a condiciones atmosféricas). Además los valores muestreados tiene validez momentánea, ya que si no se actúa dentro de un tiempo máximo estos valores no servirán.

Aún con la definición anterior algunos sistemas quedan dudosos. Podríamos decir que un cajero automático es un sistema de tiempo real porque es manejado por eventos externos como ingresar la tarjeta al cajero. El cálculo depende del momento en que se ejecuta la transacción, se tiene un máximo de retiros mensuales, existe un tiempo determinado para sacar la tarjeta o realizar operaciones. Pero una vez que se ingresó la tarjeta, el cajero automático se comporta como un sistema interactivo, ya que presenta una serie de preguntas a ser contestadas. Sin embargo tampoco es un sistema interactivo puro, porque la respuesta a dar depende del momento en que se la obtenga. Para evitar dudas provocadas por las generalidades se propone una clasificación de los sistemas de tiempo real que permite discriminar claramente entre los tipos de sistemas existentes.

Una definición más precisa llama a los sistemas de tiempo real un sistema en el cual el tiempo es el recurso mas precioso a manejar. Se deben asignar y planificar las tareas de tal forma que terminen antes que sus metas[WAI97A].

En el siguiente resumen se agrupan las distintas aplicaciones de sistemas de tiempo real existentes por el tipo de tareas que posee:

Sistemas de transporte, control de tráfico vehicular y de tráfico aéreo.
Soporte para manufactura que precisan reconfigurar rápidamente las operaciones de planta para cumplir requerimientos cambiantes, y permitir mantenimiento y actualización on-line.
Sistemas de sensores para monitorear patrones de tiempo, datos sísmicos, redes de distribución de energía, y distribución de polución.
Satélites, redes de fibra óptica y canales de alta velocidad para trasmitir audio, vídeo y texto.
Monitoreo de pacientes, tomografías computadas, pulmotores, resonancia magnética y otros equipamientos médicos avanzados.
Vigilancia, comando y control y otros sistemas de defensa.

Los sistemas de tiempo real pueden ser muy complejos o muy simples (como el microprocesador que controla una pieza de un auto).

Un sistema de tiempo real consiste de un sistema controlado y otro controlador, por ejemplo en una fábrica el sistema controlado es la planta y el sistema controlador es la computadora y los operarios

que la manejan. El sistema controlador interactúa con su ambiente basado en la información que recibe a través de sensores. Por ello es indispensable que la imagen que percibe este sistema refleje fielmente la realidad. Para ello se necesita un monitoreo y un procesamiento periódico de la información recibida.

Como estos sistemas afecta al medio ambiente, son fundamentales los requerimientos de correctitud en el tiempo. Las restricciones de tiempo pueden ser muy complicadas. Las más comunes para las tareas son restricciones de periodicidad. Una tarea aperiódica tiene un momento determinado (meta) en la cual tiene que finalizar o comenzar, o puede tener una restricción tanto en el tiempo de comienzo como de finalización. Las tareas periódicas tienen una serie continua de invocaciones regulares (el período de la tarea) y un tiempo máximo de cálculo. Son típicamente las tareas que procesan información de los sensores y las restricciones de tiempo generalmente las impone el ambiente con el cual interactuan. Algunas tareas periódicas pueden existir durante toda la vida del sistema y otras ser activadas por algún evento del ambiente y durar un tiempo determinado (ejemplo: la tarea que monitorea un avión en el radar cuando se acerca a un aeropuerto, es periódica mientras el avión está al alcance del radar). Pueden surgir requerimientos de tiempo aperiódicos de eventos dinámicos del medio ambiente (un operador presiona un botón).

La perdida de las restricciones de tiempo se debe evaluar dependiendo de la naturaleza del sistema. Por ejemplo, no es lo mismo que se pierdan metas en un sistema que controla un misil que en el monitoreo por radar de un avión acercándose a un aeropuerto. Esto nos lleva a una definición de funcionamiento correcto del sistema dependiendo de las restricciones de tiempo.
Los sistemas de tiempo real pueden dividirse en tres categorías[WAI93]:

- Sistemas de Tiempo Real "Duro": los cálculos siempre deben terminarse en un tiempo máximo especificado.
- Sistemas de Tiempo Real "Blando": el sistema debe tener un tiempo medio de ejecución tal que si lo medimos sobre un intervalo de tiempo definido, es inferior a un máximo especificado.
- Sistemas de Tiempo Real "Firme". En ésta categoría se incluye Sistemas de Tiempo Real duros en los que se pueden tolerar pérdidas, si es que existe una probabilidad baja de que éstas ocurran.

Como resultado obvio, la primera categoría tiene una restricción mucho más severa de tiempo que la segunda. En un sistema de tiempo real Blando se puede llegar a aceptar un tiempo de respuesta

variable en función de la carga del sistema (porque esta variación no representa grandes problemas para el usuario). En cambio, en los sistemas de tiempo real Duro, si la computadora no determina un resultado en un tiempo específico, puede ser extremadamente crítico porque el resultado que se está calculando puede no ser útil si se entrega tarde. El sistema debe satisfacer restricciones de tiempo de respuesta explícito (limitado), a consecuencia de riesgos severos.

Un ejemplo de sistema de tiempo real "Blando" es un sistema de atención de averías telefónicas (114) en el cual el tiempo medio de respuesta de una transacción debe ser menor que un tiempo estipulado por el ente regulador, aceptando una demora mínima sobre el promedio. Se considera de tiempo real "Blando" porque se especifica un tiempo medio de ejecución. También podemos ver de esta manera a los sistemas de atención bancaria (cajeros) o los punto de venta de pasaje de las aerolíneas.

El sistema de monitoreo de una planta nuclear se puede considerar como un ejemplo de un sistema de tiempo real "Duro". El sistema que monitorea el núcleo debe responder en menos de 1 ms ante un sobrecalentamiento. El sistema debe dar una respuesta en un tiempo finito y determinado. Otro ejemplo es una cámara de secado de algodón.

Si se puede tolerar que la cámara de algodón mencionada anteriormente perdiera un pequeño porcentaje de la producción, , estaríamos frente a un sistema de tiempo real "Firme" .

Un mal concepto ampliamente difundido sobre los sistemas de tiempo real es que deben ser rápidos. El ser rápido es una condición necesaria pero no suficiente. El sistema debe cumplir metas especificas y el ser rápido no le garantiza que las cumpla.

Existen otras dos formas de clasificar a los sistemas de tiempo real. Una de ellas es según su relación entre las escalas de tiempo de los eventos externos y las funciones ejecutadas por la computadora. Según se dé esta relación, se dirá que el sistema esta :

- Basado en el reloj, si la relación entre la computadora y los eventos externos esta dada por el paso del tiempo
- Basado en eventos, si la relación entre la computadora y los eventos externos esta dada por la ocurrencia de un evento externo

Un ejemplo de un sistema de tiempo real basado en el reloj, es un sistema con tareas periódicas, en el cual una señal equidistante en el tiempo inicia las actividades.

Un ejemplo de un sistema basado en eventos son los sistemas que se activan, cuando se cierra o abre una llave o cambia de estado un sensor, por ejemplo un sistema de alarma, cuando el sensor indica que se paso de la medición límite.

Otra clasificación importante es de acuerdo a la integración con el sistema físico, podemos clasificar a los sistemas de tiempo real como:

- Embebidos. Se usan para controlar hardware especializado en el cual se instala el sistema de computación.
- No embebidos, orgánicos. Son los sistemas completamente independientes del hardware en donde corren.
- No embebidos, débilmente acoplados. Son los sistemas que con pocos cambios pueden ejecutar en distintos hardware.

Un ejemplo de sistema de tiempo real embebido es el microprocesador que controla la mezcla de aire y combustible en los autos.

Desde el punto de vista de las funcionalidades que debe proveer un sistema en tiempo real se destacan:

- Monitoreo: obtención de información del estado del entorno físico del sistema
- Control: realización de cálculos necesarios para permitir controlar el proceso de acuerdo a los valores leídos
- Actuación: alterar el estado del entorno físico (esta función como la de monitoreo se puede realizar por medio de dispositivos de interfaces como conversores AD/DA, líneas de entrada/salida digital, generadores de pulso, etc.)

En algunos sistemas las funciones de monitoreo y actuación no existen. En otros no se monitorean variables de ambiente sino que solo tienen restricciones de tiempo. Pero se puede considerar que todo sistema de tiempo real tiene un sistema de control y un sistema controlado. Por ejemplo en una fábrica el sistema de control se puede ver como las tareas administrativas y de coordinación entre los diferentes procesos, el sistema controlado es el entorno en el cual interactúa la computadora. El sistema de control usa información obtenida del mundo real, por lo tanto los valores que utiliza este sistema deben ser consistentes con el estado real del entorno, si no los resultados no serán precisos.

Todo sistema de control dispone de un conjunto de dispositivos de entrada/salida conectados con el entorno que permiten la interacción con el sistema físico a controlar. El software que los maneja mantiene una imagen del entorno. Para que el sistema de control pueda lograr sus objetivos, esta imagen debe actualizarce a intervalos específicos con las entradas obtenidas en el entorno. La tarea de control utiliza los datos de la imagen y como resultados de sus cálculos, la actualiza. Los drivers de entrada y salida deben trasmitir los datos desde y hacia el entorno. El tiempo que tome este proceso esta determinado por las características de cada aplicación controlada. En la figura nro. 1se puede ver esta interacción.

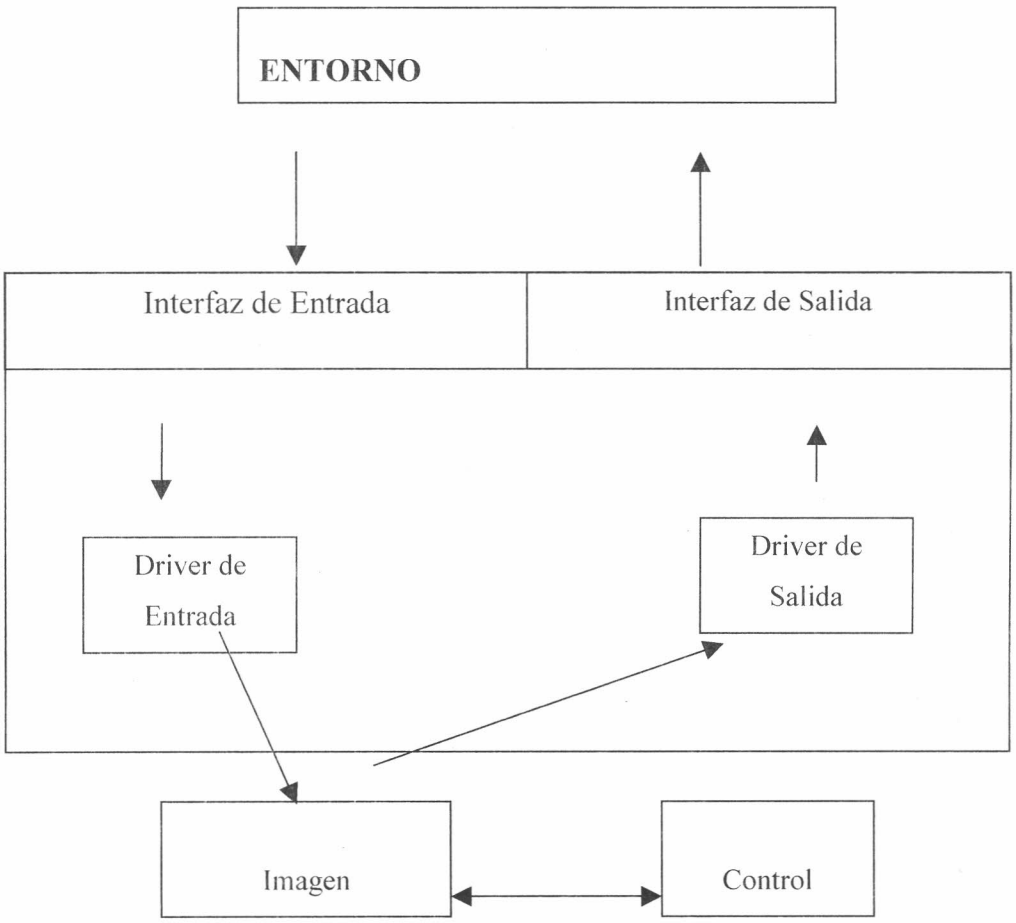


Figura 1 - Sistema de control

La adaptabilidad es muy importante en los sistemas de tiempo real, porque si una meta de una tarea puede cumplirse sólo bajo una configuración del sistema, la confiabilidad y el desempeño pueden verse comprometidos. Si un sistema es adaptable, no se tiene que redefinir el sistema o recalcular la asignación de recursos y tareas para cada cambio pequeño. Esto reduce los costos de desarrollo y mantenimiento. Dos necesidades que se derivan de la adaptabilidad, son la mantenibilidad y la expansiblidad.

Otra medida que es significativa es el time-loading o factor de carga del procesador, que mide el porcentaje de procesamiento útil que está haciendo la computadora. En los sistemas cargados al 100% no se pueden hacer cambios o agregados sin riesgo de sobrecarga. Los sistemas con baja carga tampoco son buenos porque implica que se están desperdiciando recursos.

Lograr que un sistema de tiempo real tenga las características mencionadas previamente suele resultar muy complejo, y muchas veces la organización del hardware aumenta aún más la complejidad, ya que las computadoras, sensores y actuadores pueden estar separados y funcionando en ambientes aislados. El control mismo del sistema puede estar compartido entre varias computadoras, no necesariamente todas en el mismo sitio y, por lo tanto, hay que transmitir información entre las mismas, cumpliendo las metas.

Otro punto a tener en cuenta es el sistema operativo que se usará en los sistemas de tiempo real. Este debe proveer soporte básico para satisfacer las restricciones de tiempo, tolerancia a fallas, predictibilidad y debe integrar las restricciones de tiempo con las de alocación de recursos y planificación Debe proveer facilidades para una variedad de formas de E/S que no tiene nada que ver con las E/S de los sistemas comunes por ejemplo: lectura de sensores, manejo de actuadores, comunicación con PLC, etc. También se deben tener en cuenta tres características científicas mayores [STA92]:

- Se debe elevar la dimensión del tiempo a un principio central del sistema. La mayoría de las especificaciones de los sistemas esta basada en abstracciones (ignorando los detalles de la implementación), en los sistemas de tiempo real las restricciones están dadas por el ambiente y la implementación y el éxito del sistema depende de que las cumpla, por lo que no puede abstraerse del tiempo.
- Se debe cambiar el paradigma básico de los sistemas distribuidos de propósito general. Actualmente se basa en la noción que la tarea de aplicación requiere los recursos como si los

procesara de manera aleatoria, por lo tanto se diseñaron a los sistemas operativos para que espere entradas aleatorias y tenga un buen comportamiento en el promedio de los casos. El nuevo paradigma debe estar basado en un balance entre predictibilidad y flexibilidad, este ambiente debe ser tan flexible que permita un ambiente altamente dinámico y adaptativo y debe poder predecir y prevenir conflictos de recursos para lograr que se cumplan las restricciones de tiempo. Esto se torna un poco complicado en los sistemas operativos distribuidos donde el código del sistema operativo y los protocolos de comunicación interfieren con la predictibilidad.

- Se necesita una visión muy integrada de restricciones de tiempo y alocación de recursos, así se podrán tener en cuenta correctamente las restricciones de tiempo, la predictibilidad, adaptabilidad, correctitud, seguridad y tolerancia a fallas. Los recursos deben estar disponibles a tiempo para no interferir con las restricciones de tiempo y precedencia de las tareas. Se necesita mucha coordinación para lograr un algoritmo que coordine estas dos restricciones.

Hoy en día existen muchos sistemas operativos ad hoc que generalmente están soportados por una optimización de un sistema distribuido. Para que un kernel pueda soportar los mencionado anteriormente se deben cumplir con las siguientes características[STA92]:

- Rápido cambio de contexto
- Tamaño pequeño con la mínima funcionalidad asociada
- Rápida respuesta a interrupciones externas
- Mínimo intervalo de deshabilitación de interrupciones
- Proveer particiones para el manejo de memoria de tamaño fijo y variable
- Bloqueo de código y datos de memoria
- Archivos secuenciales que puedan acumular datos a grandes tasas
- Mantener un reloj interno de tiempo real
- Proveer mecanismos de planificación por prioridades
- Alarmas especiales y tieme-outs
- Primitivas para demorar la ejecución por un periodo fijo de tiempo
- Primitivas para detener y continuar la ejecución

En computación de tiempo real se definen estas características para que sean rápidas, pero la rapidez es relativa y no es suficiente cuando se trabaja con restricciones de tiempo. Muchos diseñadores de sistemas de tiempo real creen que estas características son suficientes para un kernel

de tiempo real, en cambio otros prefieren kernels que se hagan cargo de las restricciones de tiempo y la tolerancia a fallas. Pero de ambos puntos de vista se destaca la necesidad de tener predictibilidad. Esta requiere primitivas acotadas del sistema operativo, algún conocimiento de la aplicación, algoritmos de planificación propios y un punto de vista basado en la cooperación entre la aplicación y el sistema operativo. Se necesitan algoritmos que puedan manejar la predictibilidad y la tolerancia a fallas. Se puede lograr algo construyendo algoritmos que integren la planificación del procesador y la de los recursos del modo que el conjunto de tareas pueda tener los recursos disponibles a tiempo. Otro paso importante seria que el algoritmo pudiera tomar acciones secundarias si no se puede alcanzar la meta de la tarea. (donde se puede permitir una degradación del sistema). Se debe tener en cuenta que el kernel también pueda trabajar en un ambiente altamente cooperativo de multiprocesamiento y distribución. Es decir que el kernel pueda soportar un requerimiento de tiempo de punta a punta, que un conjunto de tareas que se comunican deban completarse antes de una meta. Se esta trabajando en diseñar protocolos de comunicación con restricciones de tiempo. Si se siguen tratando estos eventos como si ocurrieran al azar no se logrará la predictibilidad deseada.

4.2. Benchmarking

Un benchmark es un test que mide la performance de un sistema o sub-sistema en un conjunto de tareas. Los benchmarks se usan para predecir la performance de un sistema desconocido sobre una tarea o carga de trabajo. También se utilizan como herramientas de monitoreo o diagnóstico. A través de estos tests y del estudio de sus resultados, se puede llegar a conclusiones sobre baja performance o impacto en la misma debido a modificaciones en el sistema.

Uno de los benchmarks usados es el Wheststone, que es un benchmark sintético, lo que significa que en vez de hacer una función útil, está construido sobre la base de la frecuencia de instrucciones de un conjunto representativo de programas. Se basa en cálculos numéricos científicos y en su mayoría son operaciones de punto flotante. Su importancia como benchmark esta basada en su amplia aceptación y uso. Otro benchmark sintético es el Dhrystone, que intenta reflejar las características de los lenguajes de programación modernos (tipos de datos de registros y punteros) e intenta ser un sistema de programación típico. El programa es una síntesis de la frecuencia de sentencias estáticas y dinámicas recogidas de 16 estudios diferentes que analizaron programas largos y cortos en una gran variedad de lenguajes.

Los benchmarks sintéticos tienden a ser mejor aceptados que los programas individuales como la ordentación (Thieve of Erathosthenes), programas recursivos (funciones de Ackermann) o juegos (Ocho Reinas). Otra ventaja importante es que sus resultados se pueden expresar en una magnitud métrica (Kilo Whetstone instrucciones por segundo) más que en un simple tiempo de ejecución.

La implementación de estos benchmarks ayuda a determinar qué componente en un sistema (algoritmo, implementación del compilador, sistema operativo, componente de hardware) es el más crítico para mejorar la construcciones de un sistema. Si se mantienen las constantes del sistema mientras se varía el algoritmo, se obtiene una comparación directa de los algoritmos. Si se mantiene constante el algoritmo y se varía la carga de los benchmarks, se puede observar la estabilidad del algoritmo bajo carga. Si se mantiene constante el algoritmo y se modifica la implementación del lenguaje, se puede comparar la performance de varios compiladores para ese lenguaje. Entonces, lo que se debe definir son los requerimientos para una aplicación sintética, no el diseño ni la implementación de una solución para estos requerimientos. El éxito de esta familia de benchmarks dependerá de su aceptación y uso.

La mayoría de los test preguntan por cuestiones de funcionalidad (se provee determinada función), capacidad (cuan grande pueden ser algunos de los aspectos de los programas), o perfomance (cuan rápido se pueden ejecutar algunos programas u operaciones). No hay test que chequeen si algún conjunto de actividades puede alcanzar sus metas preestablecidas. La respuesta a esta pregunta tiene que ver con la habilidad de mantener con exactitud el tiempo y proceder de una manera predecible y determinística más que con el througthput del sistema. Algunos sistemas de software se deben evaluar por su comportamiento predecible y determinístico mas que por su througthput.

El benchmark Rhealstone es un esfuerzo reciente para encarar la tecnología de evaluación de los sistemas de tiempo real. Se propuso definir una métrica obtenida de la suma pesada de seis categorías de actividades que se juzgaron ser las más cruciales en los sistemas de tiempo real. Estas categorías son:

- Tiempo de intercambio de tareas
- Tiempo de desalojo
- Tiempo de latencia de interrupción

- Tiempo de confusión de semáforos
- Tiempo de romper los deadlock
- Tiempo de throughput de los datagramas

La idea es usar la suma pesada como el indicador de operaciones cruciales realizadas en un segundo. Este benchmark puede ser una contribución importante pero como muchos benchmarks de grano fino tiene sus limitaciones. Aún bajo las mejores circunstancias es arriesgado tratar de predecir el comportamiento de tiempo real de un sistema basado en esta información. Estas categorías no solo que están sujetas a muchas fuentes de variación, sino que es muy difícil de calcular las operaciones en aplicaciones reales, combinando estas dos cosas puede tener implicaciones de perfomance inexplicables.

Las aplicaciones de tiempo real tienen restricciones de tiempo como parte de su especificación. Existen investigaciones importantes en las técnicas de planificación, lenguajes, sistemas operativos y bases de datos de tiempo real. La tecnología de evaluación disponible hasta el momento no está orientada a medir las restricciones de tiempo características de estos sistemas.

4.2.1. Hartstone

Hartstone es un benchmark usado para medir varios aspectos de un sistema de tiempo real duro. Se compone de una serie de requerimientos para testear la habilidad que tiene un sistema para manejar aplicaciones de tiempo real.

El término "Hartstone" deriva de tiempo real duro (Hard Real Time) y de que la carga computacional de trabajo se basa en un benchmark sintético con el mismo espíritu del Wheststone y el Dhrystone, Su definición posibilitará comparar distintas arquitecturas (hardware /software /algoritmos).

Las aplicaciones de tiempo real duro ("hard") deben cumplir sus metas para satisfacer los requerimientos del sistema, en contraposición a los aplicativos de tiempo real blando ("soft") donde se aceptan tiempos de respuesta de distribución estadística.

La mayoría de los test plantean las cuestiones de funcionalidad (se provee una función), capacidad (cuán grande pueden ser algunos aspectos de los programas), o perfomance (cuán rápido se pueden ejecutar algunos programas u operaciones). Ninguno chequea si algún conjunto de actividades puede alcanzar sus metas preestablecidas. Las respuestas a las preguntas anteriores tiene que ver más con el througput del sistema que con un conducta determinística y predecible necesaria para los sistemas de tiempo real.

Kar y Porter [WEI90] definieron una suma pesada de seis categorías de actividades que se juzgaron cruciales en los sistemas de tiempo real. Estas categorías son: tiempo de intercambio de tareas, tiempo de desalojo, tiempo de latencia de interrupción, tiempo de confusión de semáforos, tiempo requerido para romper los deadlocks y tiempo de througput de datagramas. Se usa la suma como un indicador del número de operaciones cruciales realizadas en un segundo. El problema es que como muchos benchmarks de grano fino tiene sus limitaciones aún bajo las mejores condiciones, es muy arriesgado tratar de predecir comportamiento de tiempo real basándose en esta información. Estas medidas están sujetas a muchas fuentes de variaciones y es muy difícil calcular las operaciones en aplicaciones reales, estos dos factores hacen que no se obtengan resultados fiables.

Es de gran ayuda para la evaluación de sistemas de tiempo real una familia representativa de requerimientos de benchmarks sintéticos. La implementación de estos requerimientos ayudarán a determinar qué componente en un sistema (algoritmo, implementación del compilador, sistema operativo, componente de hardware) es el más crítico para mejorar la construcción de sistemas de tiempo real . Si el éxito del Dhrystone o el Wheststone indican algo, entonces se está por buen camino.

Se deben definir los requerimientos para una aplicación sintética, no se debe definir el diseño ni la implementación de una solución para esos requerimientos solo definir los conceptos operacionales de una serie de requerimientos de benchmark a ser usados para testear la habilidad de un sistema para trabajar con aplicaciones de tiempo real. [WEI89]

Los requerimientos del benchmark Hartstone definen una serie de test. Los requerimientos iniciales de una serie dada proveerán una base de donde se obtendrán los demás. Cada test de la serie puede tener éxito (llegar a sus metas de tiempo real) o fallar (perder una o más metas durante el test). Entre las características de los requerimientos que definieron se encuentran las siguientes:

- Medir el dominio del problema de tiempo real duro: los requerimientos deben ser representativos del dominio del problema para el cual es diseñado. Las aplicaciones de tiempo real duras están caracterizadas por actividades periódicas, procesos aperiódicos generados por interrupciones o la interacción del usuario, sincronización entre las actividades, acceso a datos comunes, cambios de modo, y procesadores distribuidos. Se requiere que Hartstone sea tan parecido como sea posible a las aplicaciones actuales del mundo real. Sin embargo, alguna de estas características no serán incorporadas en los requerimientos simples.

- Complejidad creciente: Los requerimientos se irán incrementando en complejidad. Se presume que se irán implementando desde los simples a los complejos.

- Testeo de Estrés: Cada test individual será estructurado para tener un requerimiento base y una estrategia para modificar ese requerimiento a forzar al sistema a sus límites a través de un número de dimensiones. Estas dimensiones pueden incluir, por ejemplo, la carga del procesador, la cantidad de tareas periódicas y aperiódicas en actividad, y la frecuencia en que se activan. Cada familia de test tendrán una "figura de mérito" que dará una indicación de la cantidad de trabajo útil realizado durante el test.

- Auto - verificación: Cada test individual debe verificar que los cálculos se realizan correctamente y que se alcanzan las metas.

- Trabajo de carga sintético: Una carga sintética garantiza que se está haciendo una cantidad igual de trabajo en cada sistema corriendo a los benchmark independientes de hardware y software. La carga sintética debe ser representativa del trabajo que se debe hacer en los sistemas de tiempo real. La cantidad de trabajo debe ser variable para facilitar a los programas de benchmark que se acerquen más a los diferentes ambientes de aplicación.

- Figura de mérito relativa: La métrica que indica los resultados de la serie debe ser relativa mejor que absoluta, que distinga claramente la ejecución de la carga de trabajo productiva del overhead del planificador, intercambio de tareas, sincronización de tareas, mantenimiento del tiempo, y manejo de interrupciones. Es más importante conocer la máxima ejecución realizable antes de comenzar a perder las metas, que conocer el throughput del sistema. La figura de mérito relativa se puede usar para estimar el throughput para sistemas largos y cortos.

Se tiene un gran número de buenas razones para elegir tanto a Whetstone o Dhrystone como la carga sintética para estos experimentos:

- Están bien definidos e implementados en varios lenguajes diferentes.
- La velocidad desarrollada por el hardware y el sistema de runtime (en términos de Whetstones o Dhrystones) se puede usar para obtener el overhead del algoritmo de planificación y la implementación del planificador. El porcentaje del throughput desarrollado indicará cuánto tiempo se pierde debido a los diferentes overheads y bloqueo.
- Cada miles de Whetstones o Dhrystones están en el orden de los pocos milisegundos para la mayoría de la máquinas (generalmente menor a los 10 milisegundos). Esto hace posible considerar frecuencias superiores a los 100 Hz para una sola actividad que requiera 100 Whetstones o Dhrystones.
- Los benchmark Whetstones y Dhrystones son instrucciones mezcladas que están aceptadas en la comunidad como representativas de un amplio espectro de aplicaciones de computación. Los Whetstones son característicos de la programación científica, y los Dhrystones de los sistemas de computación.

Se decidió usar a los Whestones como base para los Hartstones porque son más representativos de la naturaleza numérica y científica del dominio del problema. Se obtuvo una versión revisada de los Whetstones del Laboratorio Nacional de Física, que se autovalidan y permite correr a 1 Kilo Whetstone de instrucciones (KWI) por tiempo [WEI89].

Se delinearon los requerimientos para cinco series de test y se definieron en detalle los requerimientos de una serie. Se trató de proveer una descripción para las serie definida en detalle, que sea lo suficientemente especifica para que no haya ambigüedades de lo que se esta implementando, pero lo suficientemente genérica para que la implementación se pueda realizar en una variedad de lenguajes y en una variedad de sistemas.

Para las descripciones generales, se trata de proveer la noción de extender los requerimientos iniciales para que sea más aplicable a los sistemas de tiempo real. En todos los casos, el diseño de la solución debe ser completamente dependiente del sistema.

Se definen las siguientes categorías de testeo en orden creciente de dificultad

Series PH: tareas periódicas frecuencias armónicas: El objetivo de esta serie es el de proveer los requerimientos del test con un conjunto de tareas que son periódicas y armónicas. Esto significa que cada tarea en el sistema corre a intervalos regulares precisos, y la frecuencia de cada tarea es un múltiplo integral de todas las tareas con frecuencias bajas. Por ejemplo, un conjunto de tareas con cuatros tareas corriendo a frecuencias de 1 Hz, 5 Hz, 10 Hz, y 20 Hz puede ser un conjunto de testeo armónico. La importancia de las frecuencias armónicas es que se encuentran ampliamente en las aplicaciones de tiempo real y son las más fáciles de manejar usando un número de diseños y algoritmos de planificación. Estas series pueden representar un programa que monitorea varios bancos de censores a diferentes tasas y muestra los resultados sin la intervención de usuario ni requerimientos de interrupción. Se podría implementar como un ejecutivo cíclico y tendrá la utilización teórica más alta (100%) para un planificador de Tasa Monotónica.

- Series PN: tareas periódicas, frecuencias no armónicas: El objetivo de esta serie es el de proveer los requerimientos del test con un conjunto de tareas que son periódicas y pero que no son armónicas. Estas series representan un dominio de aplicación similar al de las series PH, pero en el cual las frecuencias están elegidas para que concuerden con los requerimientos de la aplicación (frecuencias naturales de fenómenos físicos) más que con los requerimientos de implementación (las frecuencias requeridas por el hardware o los detalles de implementación). Este conjunto de tareas no esta particularmente responsabilizado de un diseño cíclico y tiene muy poca utilización en la teoría de planificación (el nivel de utilización en el cual el planificador de Tasa Monotónica puede garantizar las metas) [LIU 73].
- Serie AH: Esta serie es la serie PH con el agregado de procesamientos aperiódicos. Su objetivo es el de proveer los requerimientos del test con un conjunto background de tareas que son periódicas y armónicas y un conjunto foreground de tareas aperiódicas manejadas por interrupciones. Esta serie representa un dominio de aplicación en el cual el sistema responde a eventos externos. Por ejemplo, los sistemas de control de combate y las interfaces de usuario son frecuentemente manejadas por interrupciones. La tarea aperiódica puede estar caracterizada por intervalos de tiempo que se toman de distribuciones estadísticas y pueden tener o no metas. Es importante minimizar los tiempos de respuesta mientras se sigan cumpliendo todas las metas del conjunto de tareas background. Se propusieron recientemente un número de algoritmos para aumentar el tiempo de respuesta en este contexto. Este conjunto de tareas se puede representar tanto por ejecutables cíclicos como por ejecutivos manejados por datos.
- Series SH : es la serie PH con sincronización. Su objetivo es el de proveer los requerimientos de test con un conjunto de tareas que requieren sincronización entre ellas. La sincronización

introduce la posibilidad del bloqueo de tareas y un conjunto de situaciones de inversión de prioridades. Este conjunto de tareas se puede usar para testear la efectividad de varios algoritmos y protocolos que tratan con la inversión de prioridades. También seria útil investigar la eficiencia de varios mecanismos de sincronización.

- Series AS: es la series PH con procesamientos aperiódicos y sincronización. Su objetivo es el de proveer los requerimientos de test con un conjunto de tareas que combinan todas las características de los test anteriores. Esta serie es la más compleja y la que más demanda a un sistema para manejarla. Contendrá tareas periódicas y aperiódicas como también sincronización entre las tareas. Esta serie de test es lo suficientemente compleja que permite que se le modifique para prototipar a muchos otros sistemas reales. Proveerá un buen testeo del sistema y muchas de las características requeridas es los sistemas de tiempo real.

El conjunto de tareas para la serie PH ahora puede ser la base para un número de experimentos que testean la sensibilidad del runtime del sistema para un número de cambios diferentes. La secuencia del test puede ser la base para una métrica que sea indicativa de como soporta el sistema los test de conjunto de tareas armónicas.

Se definen 4 experimentos como sigue:

- Experimento 1: La frecuencia de la tarea de mayor frecuencia se incrementa en una cantidad hasta que se pierda una meta. Esta secuencia incrementa el trabajo de carga total y testea la habilidad del sistema para manejar una granularidad fina de tiempo y la rapidez del cambio de contexto entre procesos.
- Experimento 2: Comenzando con el conjunto inicial de tareas, se escalan todas las frecuencias por 1, luego por 2, después por 3, y así hasta que se pierda una meta. Esta secuencia incrementa el trabajo de carga total y testea la habilidad del sistema para manejar un trabajo de carga incremental pero balanceado.
- Experimento 3: Comenzando con el conjunto inicial de tareas, el trabajo de carga de cada tarea se incrementa escalándolo por 1, luego por 2, después por 3 unidades por periodo, y así hasta que se pierde una meta. Esta secuencia incrementa el trabajo de carga total sin incrementar significativamente el overhead del sistema.
- Experimento 4: Comenzando con el conjunto inicial de tareas, se agregan tareas nuevas al conjunto hasta que se pierde una meta. Esta secuencia testea la habilidad del sistema para maneja un gran conjunto de tareas.

Los programas de Hartstones ejercitan las características importantes del comportamiento del runtime, incluyendo el intercambio de tareas, el manejo de interrupciones, rendezvous, el comportamiento del plan, y el determinismo del sistema de runtime. Se determinó que los programas Hartstone brindan a la luz los comportamientos inesperados de los problemas de perfomance que pueden ser investigados con programas de benchmarks de grano fino. Para interpretar correctamente los resultados, el usuario debe poseer conocimiento detallado del tiempo de ejecución de las operaciones del reloj, rendezvous de las tareas, cambios de contexto, latencia de interrupción de timer, interacción con el coprocesador de punto flotante (si hay), y el overhead del llamado a los procesos. Además de proveer una vista interior de las características de perfomance de tiempo real de un sistema de runtime, los programas Hartstone proveen también ejemplos canónicos del procesamiento de tiempo real que pueden ser útiles para el modelado y la experimentación.

La elección del benchmark Hartstone para poder comparar los algoritmos de planificación del procesador implementados en la primera etapa del presente trabajo es la más natural. Dado que Hartstone no es un benchmark en sí, si no que es un conjunto de requerimientos de benchmark para ayudar en el estudio de los sistemas de tiempo real y que es específico para tiempo real hacen de esta elección, la más natural.

Al ser un conjunto de requerimientos se puede independizar tanto del sistema operativo como del lenguaje para construir esta herramienta. Otra ventaja es que el sistema de métrica que se utiliza no depende del hardware en donde se esta probando, con lo que lo hace muy conveniente para los sistemas de tiempo real ya que estos se implementan en una variedad de hardware, permitiendo diseñar una herramienta de comparación de algoritmos muy útil.

4.3. Planificación para Tiempo Real

La planificación involucra la alocación de recursos y tiempo a las tareas de forma tal que se cumplan ciertos requerimientos de performance. El problema básico en sistemas de tiempo real es asegurar que las tareas cumplan sus restricciones de tiempo [WAI94].

Se debe planificar la utilización de los recursos del sistema de tal forma que su comportamiento temporal sea comprensible, predecible y mantenible. Para ello el sistema operativo debe sincronizar

cuidadosamente las diversas tareas en ejecución. Ya no es prioritaria la optimización del uso de los recursos del sistema, sino la obtención de respuestas correctas en un tiempo máximo determinado [WAI94].

La restricción de tiempo indica cuán urgente es una tarea, pero esto puede no tener relación con lo crítica que puede ser dicha tarea. Por este motivo la planificación de tiempo real es un problema no trivial ya que no solo las tareas se deben programar para que cumplan sus restricciones de tiempo sino que también se debe dar prioridad a las más críticas [WAI94].

El objetivo de un planificador de tiempo real será verificar si existe un plan para el conjunto de tareas. El patrón que se utiliza para medir el mérito de estos planificadores es distinto al de los planificadores de sistemas tradicionales. Los criterios como rendimiento, tiempo de retorno, grado de utilización del sistema o tiempo medio de respuesta no indican nada, se tiene que considerar [WAI95b]:

- repuestas predecibles ante eventos externos: hay que asegurar que las respuestas se obtengan antes de una metas dada. Al activar una tarea se debe poder determinar correctamente su tiempo de finalización tomando en cuenta todos los factores que puedan obstaculizar su finalización (estado del sistema, recursos que necesita). Por ello hay que conocer las condiciones extremas de carga del sistemas antes de su ejecución.
- Alto grado de planificabilidad cumpliendo los requerimientos de tiempo de las tareas. O sea debe tratarse que la utilización de los recursos sea optima, trate de ejecutar la mayor cantidad de tareas predeciblemente.
- Estabilidad ante sobrecargas momentáneas. Cuando el sistema esta sobrecargado se debe garantizar el cumplimiento de las tareas más criticas.
- Confiabilidad, se debe garantizar el correcto funcionamiento en todas las situaciones, dado que este tipo de sistemas se suelen usar en sistemas críticos.
- Adaptabilidad a cambios de configuración, especificaciones y estado del sistema. Cuanto mas adaptable sea el sistema será mas extensible y mantenible.

No se puede asegurar que un conjunto de tareas 100% planificable no pierda metas, ya que puede ocurrir un error de planificación, dado que la mayoría de los análisis están basados en suposiciones no muy reales. Si lo que se asume no esta de acuerdo con el ambiente, entonces el análisis carece de valor. La idea es elegir un algoritmo que lo que asuma este lo mas cerca posible de la realidad. El

problema es que cada vez se trata con ambientes mas complicados, y en la mayoría de los algoritmos que se han presentado no se han tenido en cuenta muchas características realistas[STA92] como las siguientes:

- Tareas con y sin desalojo
- Tareas periódicas y aperiódicas
- Tareas con niveles de importancia múltiples o que dependan de una función
- Grupos de tareas con una sola meta
- Restricciones de tiempo punto a punto
- Restricciones de precedencia
- Requerimientos de comunicación
- Requerimientos de recursos
- Restricciones de alocación
- Necesidades de tolerancia a fallas
- Metas cerradas y relajadas
- Condiciones de carga normales y sobrecarga
- Integrar la planificación de la CPU y la alocación de recursos
- Integrar la planificación de E/S y la planificación de la CPU
- Integrar la planificación de la CPU y la planificación de tiempo real de las comunicaciones
- Integrar la planificación local y distribuida
- Integrar la planificación estática de las tareas críticas y la planificación dinámica de las tareas esenciales y las no esenciales.

Realizar un planificador con todos estos factores en mente es una tarea muy complicada. Algunos algoritmos relajan algunos de estos factores para que el mismo no sea tan complejo. Para ello se consideran algunos factores genéricos existentes en los sistemas de tiempo real.

Muchos planificadores consideran solo dos tipos de tareas: periódicas y aperiódicas o esporádicas. Las tareas periódicas son las más comunes ya que por lo general los sistemas de tiempo real incluyen algún tipo de muestreo regular. Estas tareas tienen una serie continua de invocaciones regulares y un tiempo máximo de cálculo. Las tareas aperiódicas se ejecutan una sola vez cuando son invocadas y su activación y duración no registra ningún patrón [WAI95B].

Muchos algoritmos de planificación para estos sistemas son estáticos, el algoritmo tiene completo conocimiento del conjunto de tareas a planificar (metas, tiempos de cálculo, precedencia, etc.)[STA94], este algoritmo produce un plan fijo y puede determinar la secuencia de activación antes que comience a ejecutar el conjunto de tareas. En este tipo de planificación se le da una prioridad fija a cada tarea y se puede realizar manualmente el plan. Este tipo de planificación no es la ideal al ser muy rígida, pero es realista para muchos sistemas de tiempo real (un experimento simple en un laboratorio o una aplicación simple de control que tiene un conjunto fijo de sensores y actuadores y un ambiente bien definido y acotado). Los algoritmos de planificación dinámicos conocen el conjunto de tareas inicial pero admiten que durante la ejecución lleguen nuevas tareas de las que no se tenia conocimiento, entonces la planificación cambia en el tiempo. Este tipo de algoritmos se requieren en aplicaciones tales como un conjunto de robots sincronizados para realizar alguna tarea. La planificación off-line no es lo mismo que planificación estática, esta planificación es el análisis que se realiza para ver si un conjunto de tareas es planificable.

Otro aspecto de los algoritmos de planificación es el desalojo de tareas, esto es, si una tarea ya cumplió el tiempo asignado por el procesador, se desaloja del mismo para darle paso a otra tarea.

La planificación de tareas es compleja si se usan los sistemas operativos de la actualidad, ya que en muchos casos sus algoritmos de planificación son modificaciones a los algoritmos de planificación de tiempo compartido. Muchos algoritmos usan planificación por prioridades fijas y el implementador tiene que ajustar las prioridades a mano (una solución ad-hoc) que trata de garantizar el cumplimiento de las restricciones de tiempo con pruebas exhaustivas. Esta técnica implica costos elevados, gran inflexibilidad y además obliga a que dos medidas importantes como la criticidad de una tarea y su restricción de tiempo se integren en una sola medida, la prioridad[WAI95C].

El objetivo de un planificador de tiempo real será determinar si hay un plan que cumpla las restricciones de tiempo de todas las tareas que se van a ejecutar. Una tarea es planificable si existe un plan en el cual la tarea puede cumplir sus restricciones de tiempo.

Algunos sistemas de tiempo real son lo suficientemente simples y estáticos, involucran relativamente pocas tareas, que la planificación puede ser echa offline. Por ejemplo, si el conjunto de tareas consiste solamente en tareas periódicas, entonces cuando se determina la primera invocación todas las demás invocaciones se podrán determinar exactamente. El plan offline

resultante se denomina *ejecutivo cíclico* y puede ser muy eficiente reduciendo el overhead del intercambio de tareas y ofrece un método simple para forzar la exclusión mutua mientras se está accediendo a un recurso compartido[WAI95B].

Estos ejecutivos cíclicos, sin embargo, tienen algunos problemas. Si se cambia alguna tarea del conjunto se deberá volver a calcular y testear la planificación. La ejecución se complica cuando el conjunto de tareas es grande, no puede manejar tareas aperiódicas y esporádicas, para ser eficiente el periodo de las tareas debe ser armónico y no puede manejar las situaciones en donde una tarea ejecuta más de lo esperado.

Otros sistemas operativos para tiempo real usan round robin, en esta planificación cada tarea tiene asignado un tiempo de procesador (el tiempo es pequeño y se lo denomina quantum), cuando se termina el tiempo la tarea se desaloja y se asigna el procesador a otra tarea. La asignación del procesador es por orden de llegada. Esta planificación tiene el problema que si una tarea de alta prioridad hace mucho uso del procesador, esta puede ser desalojada antes que termine. Y si una tarea tiene una meta que es mayor al quantum, no se puede asegurar que la misma cumpla su meta (los sistemas operativos que usan este tipo de planificación permiten variar el quantum de tiempo, pero esto tampoco asegura que se cumplan las metas).

Una de las planificaciones tradicionales que tiene mayor utilidad con los sistemas de tiempo real es la planificación basada en prioridades externas, esto es: se selecciona la tarea con mayor prioridad entre las que están listas para ejecutar. Las prioridades de las tareas pueden ser fijas o variables. Los sistemas con prioridades fijas son menos flexibles y una vez comenzado el sistema no se pueden variar las prioridades. En cambio los sistemas con prioridades variables permiten que la prioridad de una tarea cambie durante la ejecución del sistema. Lo que puede ocurrir es que la tarea de mayor prioridad nunca abandone el procesador por lo tanto las demás tareas no podrán cumplir sus metas, ni ejecutar (esto se conoce como inanición), para prevenir esto se usa la técnica de envejecimiento en el cual la prioridad de una tarea se sube si pasa mucho tiempo sin ejecutar. Esta técnica no sirve en los sistemas de tiempo real porque la prioridad de una tarea no refleja la urgencia de ejecución.

La política de planificación por prioridades internas no es efectiva, no es de utilidad ejecutar primero la tarea más corta o la que usa más un recurso, porque esto no refleja la importancia de la tarea en el sistema, ni las propiedades temporales de las mismas.

Otra clase de algoritmo que se suele utilizar en los sistemas de tiempo real son los algoritmos multicolas. Esto es manejar distintos niveles de colas de listo que se manejen con distintas planificaciones. Cada nivel tiene una prioridad y se ejecuta primero las colas de mayor prioridad y cuando se vacía se ejecuta la de menor prioridad. Este tipo de planificación permite una asignación dinámica y eficiente de los recursos. El manejo de las tareas en cada cola será más sencillo, lo que facilitará que se cumplan las metas.

El problema de adaptar estos algoritmos que no son de tiempo real a los sistemas de tiempo real es que las soluciones terminan siendo ad-hoc. Lo que hace que el sistema no sea ni mantenible, ni flexible.

La gran cantidad de factores a considerar hace muy complejo el tema de encontrar un algoritmo de planificación para sistemas de tiempo real. Existen dos visiones para tratar el problema de la complejidad de un algoritmo de planificación. La primera visión es tratar de acotar las restricciones impuestas al conjunto de tareas (teniendo en cuenta situaciones reales donde pueden no aparecer todos los factores mencionados) y tratar de planificar al conjunto de forma determinística, esto nos lleva nuevamente al problema de no tener una algoritmo general, sino una solución ad-hoc. El otro punto de vista es la utilización de algoritmos basados en heurísticas que permite acotar la complejidad de los cálculos, pero no se puede garantizar la planificabilidad de las tareas (la heurística elegida nos puede decir que un conjunto no es planificable cuando en realidad lo es).

A continuación se analiza un conjunto de algoritmos de planificación para tiempo real enfocando sus ventajas y desventajas. Cada uno de ellos reduce la complejidad aplicando algunas de las soluciones que se mencionaros anteriormente.

4.3.1. Tasa Monotónica (TM)

El concepto de planificación de tasa monotónica fue primeramente introducido por Liu y Layland en 1973. Es el algoritmo de mayor difusión y sirve para planificar tareas periódicas independientes. El término Tasa Monotónica deriva de un método de asignación de prioridades a un conjunto de procesos: asignando prioridades como una función "monotónica" de la "tasa" de un proceso periódico. Da mayor prioridad a las tareas con periodos de ejecución más breves y las tareas de mayor prioridad pueden desalojar a las tareas de menor prioridad. Dada esta regla simple para asignar prioridades, la teoría de planificación de tasa monotónica provee la siguiente desigualdad – comparando la utilización total del procesador con una cota teórica determinada – que sirve como

condición suficiente para asegurar que todos los procesos completarán sus trabajos al finalizar sus períodos.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n.\left(2^{1/n} - 1\right)$$

Si llamamos T_i al periodo de la i-ésima tarea y C_i a su peor tiempo de ejecución.

Esta cota tiende a *ln* 2 cuando n tiende a infinito (aproximadamente el 69% del uso del procesador) [LIU73]. Este es el test básico de planificabilidad de tasa monotónica. Esta cota es muy restrictiva ya que es muy raro que ocurra el peor caso que es con el que se calcula. Este chequeo es una condición suficiente pero no es necesaria para asegurar la planificación. En [LIU73] se propone otro chequeo que es una condición necesaria y suficiente para asegurar la planificación: si una tarea cumple sus metas cuando todas las tareas del conjunto comienzan al mismo tiempo, siempre cumplirá sus metas para toda combinación de tiempos de comienzo. Una manera empírica de comprobarlo [WAI95b] es ver si la tarea cumple su primer meta cuando todas comienzan simultáneamente, como el conjunto de tareas es periódico la combinación temporal se repetirá luego del mínimo común múltiplo de los periodos de las tareas. Con este resultado se mejora la carga del procesador que ahora es del 88% y si los periodos de las tareas son armónicos (son múltiplos del periodo más corto) entonces la carga se puede mejorar al 100%[LIU73].

Se ha demostrado que es óptimo (si no existe un plan para un conjunto de tareas que no pueda resolverlo tasa monotónica entonces no existe ningún plan, ningún otro algoritmo podrá resolverlo) y estable.

El uso del procesador en este algoritmo es bajo. Esto se puede solucionar usando el algoritmo para hacer planificación on-line en vez de off-line [WAI95b]. Otra alternativa es aprovechar el tiempo ocioso para realizar tareas en background.

Este algoritmo no considera la ejecución de tareas esporádicas y aperiódicas. Una manera de solucionar este problema es realizar planificación on-line y considerar a las tareas aperiódicas y esporádicas como tareas periódicas con una única instancia y periodo igual a su meta. Otra alternativa es tratarla como una tarea periódica que hace polling (si no ocurre el evento esperado no ejecuta hasta el próximo periodo) [LUI93]. Si durante el periodo de polling hay muchos eventos la

tarea se demorará y puede ser que se pierda alguna meta o si ocurre un evento después que la tarea revisó los eventos, este evento deberá esperar hasta el próximo periodo. Para evitar estos problemas se propusieron soluciones alternativas como los servidores aperiódicos: mejorar el tiempo de respuesta de las tareas aperiódicas ejecutándolas en los tiempos libres de ejecución que dejan las tareas periódicas.

Aunque el algoritmo es estable, ante sobrecargas su prioridad es proporcional al periodo, basado en la idea de que las tareas más críticas son las de periodos más cortos, esta noción no es siempre cierta y puede pasar que una tarea crítica de periodo largo no llegue a ejecutar ante una sobrecarga, por ejemplo una tarea aperiódica de emergencia muy larga. Para este tipo de problemas se usa la técnica de transformación de periodo [LUI93] que combina la criticidad de una tarea con su periodo para que se pueda planificar por tasa monotónica. La idea es hacer como si una tarea crítica tuviera la mitad de su periodo y si se supera el tiempo de ejecución mayor a la mitad del peor tiempo esperado se suspende la tarea. Así la tarea de emergencia tendrá mayor frecuencia.

Otro problema de este algoritmo es que esta pensado para tareas independientes por lo que puede provocar inversión de prioridades en tareas dependientes, no se pueden eliminar todas las inversiones de prioridad pero sí se pueden limitar en tiempo. La inversión de prioridades se da cuando dos tareas con prioridades baja y alta respectivamente, tienen exclusión mutua y la tarea de baja prioridad bloquea la ejecución de la tarea de alta prioridad por el uso de esa zona de exclusión, además si la tarea de prioridad baja es desalojada por tareas de prioridad media, se estará retrasando mucho más la ejecución de la tarea de prioridad alta. Una solución a este problema es no permitir que se desaloje a la tarea de prioridad baja mientras esta ejecutando en una sección crítica. Una manera simple de hacerlo es deshabilitar el desalojo mientras la tarea esta ejecutando en la sección crítica. Este procedimiento es efectivo si la sección crítica más larga es mucho menor que el periodo más corto. Otra forma de evitarlo es utilizando el protocolo de herencia de prioridades en donde si una tarea de baja prioridad bloquea a otras de mayor prioridad, esta tarea hereda la prioridad más alta de las tareas que bloquea hasta que sale de la sección crítica. Este protocolo sufre de problemas de bloqueos encadenados. Se logra descartar este problema con la emulación de techo de prioridades que inhabilita selectivamente la remoción de las tareas de prioridad media.

Si bien la utilización del procesador es baja, en la práctica se prefiere este algoritmo a otros con mayor utilización por las siguientes consideraciones practicas[LUI93]: muchos sistemas de tiempo real tienen sistemas de muy baja prioridad que aprovechan los tiempos libres del procesador y

considerar la estabilidad ante sobrecarga es más importante. Cuando el sistema está sobrecargado es muy importante la garantía de poder seguir cumpliendo las metas de las tareas mas críticas. En una asignación estática solo hay que asegurarse que las tareas críticas tengan la mayor prioridad

4.3.2. Servidores Diferidos - Servidores Esporádicos

Este algoritmo nace de una modificación al algoritmo de tasa monotónica para que maneje tareas aperiódicas o esporádicas [WAI95B]. El algoritmo consta de un servidor que es una tarea conceptual con un presupuesto de ejecución y un periodo de actualización. Se calcula la cantidad de tiempo libre en cada bloque de ejecución y luego se pide que las tareas esporádicas que lleguen se ejecuten en esa cantidad de tiempo libre, dándole mayor prioridad a las tareas periódicas. Al principio de cada periodo del servidor se le asigna su presupuesto de ejecución y cuando se agota su reserva deberá ejecutar en background con una prioridad correspondiente a su máxima frecuencia de activación. Es muy complejo de manejar cuando hay muchos servidores con distintos niveles de prioridad.

Una mejora a este algoritmo es la llamada servidores esporádicos en la que se restablece el presupuesto de tiempo del servidor cuando este se ejecuta (en vez de periodos fijos de tiempo). Mientras se tenga tiempo de ejecución las tareas aperiódicas puede desalojar a las periódicas, y después se ejecutan con la prioridad original. La ventaja de este algoritmo sobre el anterior es que la carga del sistema solo se ve afectada si hay alguna tarea esporádica para ejecutar y se pueden usar los mismos teoremas de predictibilidad que para tasa monotónica dando a la tarea esporádica un periodo igual a su peor tiempo posible de activación.

4.3.3. Cálculos Imprecisos – Polimorfismo de Desempeño

Estas técnicas basadas en el algoritmo de tasa monotónica tratan de lograr la estabilidad ante sobrecargas logrando una degradación decorosa al activar rutinas alternativas ante sobrecarga. La primera técnica permite que las tareas devuelvan resultados menos precisos. Esta técnica se puede aplicar cuando la tarea que se ejecuta realiza iteraciones refinando el resultado, entonces se devuelve el resultado cuando la tarea termina normalmente o cuando se cumple la meta. La otra técnica permite activar diversas versiones de la misma función que permiten variar el tiempo de ejecución o los recursos consumidos de acuerdo a las necesidades. Entonces ante la presencia de sobrecarga se puede activar una función distinta que no es tan buena pero es más rápida[WAI95B].

4.3.4. Meta Más Corta Primero (DD)

Este algoritmo es que alcanzó más aceptación después del de tasa monotónica. Este es un algoritmo de prioridades variables con desalojo, apto para planificar tareas con tiempos de llegada aleatorios. Se le da prioridad a las tareas con metas más próximas a cumplirse. También tiene una formula de garantía que analiza el conjunto de tareas:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Se pudo demostrar con diversos estudios que en el caso de sobrecarga el algoritmo de tasa monotónica de comporta mejor que este, porque ejecuta primero las tareas más críticas logrando estabilidad, en cambio cuando se estabiliza el sistema logra mejor perfomance este algoritmo porque tiene una cota mas relajada de utilización del procesador, además favorece a las tareas con metas más cercanas reduciendo sus tiempos de respuesta, por lo tanto al introducir tareas esporádicas la situación cambia. Y la diferencia se incrementa si las tareas esporádicas tienen metas más cortas que los periodos de las tareas periódicas. Al ser un algoritmo dinámico tiene un mayor costo de invocación. Un problema típico de este algoritmo es que cuando el sistema está sobrecargado la primera tarea que pierde su meta puede causar que todas las demás la pierdan, si pasa esta situación el algoritmo de meta más corta primera no garantiza que tareas pueden llegar a cumplir sus metas. Esta condición no es alentadora para los sistemas con sobrecargas intermitentes. Este algoritmo es óptimo para los casos en que no existen bloqueos. Una solución para ejecutar tareas con restricciones es el uso de bloques de planificación mutuamente excluyentes con el mismo tiempo de cálculo[WAI95b](monitor de núcleo), se trata de asignar al procesador en un quantum de longitud mayor a la mayor sección crítica, según esta solución si existe un plan factible para un conjunto de tareas con restricciones de precedencia este algoritmo lo encuentra. La principal desventaja de este algoritmo es la falta de determinismo ante situaciones de sobrecarga.

4.3.5. Meta Predictiva

Este algoritmo además de las metas considera las estimaciones de tiempo de ejecución. Cada tarea tiene una prioridad estática y una estimación de su tiempo de ejecución que se usa para predecir cuando perderá una meta. Se asume una asignación de N prioridades estáticas donde la prioridad más baja es más crítica. Cada tarea es una unidad de ejecución con tres parámetros : meta,

estimación de tiempo de cálculo y prioridad. El algoritmo produce una lista ordenada por meta y criticidad y genera otra lista con las tareas que no podrán cumplir sus metas (para avisarles).

4.3.6. Least Laxity First (LLF)

Este algoritmo considera la flexibilidad para realizar su planificación. A las tareas con mayor flexibilidad se les asigna una prioridad más alta. Medimos la flexibilidad como la diferencia entre la meta y el tiempo restante de cómputo para terminar.

La flexibilidad nos indica el tiempo disponible para planificar una tarea. Esto quiere decir que se puede esperar esta cantidad de tiempo antes de planificar al tarea y todavía cumplir las metas. La diferencia principal entre este algoritmo y el de Meta Más Corta Primero (DD) es que en este caso se tiene en cuenta el tiempo de ejecución de las tareas. Mientras se lleva a cabo la ejecución, la flexibilidad de aquellas tareas que no se están ejecutando decrece. Este proceso continúa hasta que una de las tareas que no está ejecutando tiene la flexibilidad menor y se le da la mayor prioridad.

Si una tarea tiene flexibilidad negativa quiere decir que no cumplirá sus metas, por lo tanto este algoritmo provee detección de fallas temporales. Para que el conjunto de tareas sea planificable, la carga total del procesador debe ser menor que el 100%.[LUI73]

4.3.7. Maximum Urgency First (MUF)

Este en un algoritmo mixto. Combina las ventajas de los otros: predictibilidad bajo sobrecargas y límite de planificación del 100% para el conjunto crítico de tareas. El algoritmo necesita que el usuario defina la criticidad de una tarea (alta/baja), con esto se generan dos listas de tareas, cada lista se ordena por períodos crecientes. Las tareas listas más críticas con menor flexibilidad se eligen para correr en todo momento. Si no hay tareas críticas listas, las tareas con criticidad baja se eligen, nuevamente con una heurística de menor flexibilidad. El período de una tarea puede modificarse sin sacarla del conjunto crítico, si su carga (Ci/Ti) permanece sin cambios. Esto es útil en entornos dinámicos, donde el sistema necesita flexibilidad para adaptarse a situaciones cambiantes.

5. EXTENSIONES A MINIX 1.5

5.1. Minix

Minix. es un sistema operativo multitarea compatible con Unix (Versión 7). A diferencia de Unix, cuyo kernel es un programa monolítico que no esta dividido en módulos [TAN88], Minix es un conjunto de procesos que se comunican entre sí y con programas de usuarios mediante una sola primitiva de comunicación: la transmisión de mensajes. Este diseño ofrece una estructura más modular y flexible.

Esta estructurado en cuatro capas donde cada una de ellas realiza una función bien definida.

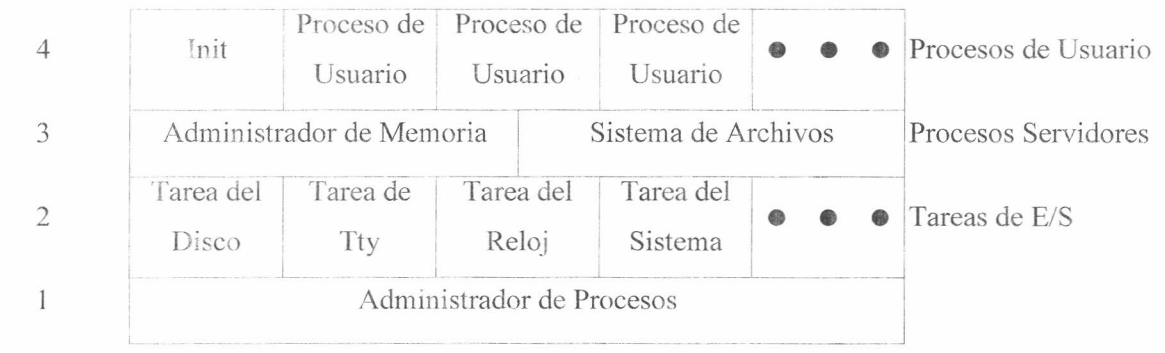


Figura 2 – Estructura en capas de Minix

La primera capa captura las interrupciones y proporciona a las capas superiores un modelo de procesos secuenciales independientes que se comunican mediante el uso de mensajes. Esta capa debe capturar las interrupciones, guardar y restituir registros, etc., en otras palabras hace la abstracción del proceso para que puedan trabajar las capas superiores y debe manejar la mecánica de los mensajes; verificación de los destinos legales, localización de los buffers (emisor y receptor) y la copia de bytes del emisor al receptor. La parte de la capa que trata con el nivel mas bajo de manejo de interrupciones está escrito en assembler, lo demás está escrito en C.

La segunda capa contiene los procesos de E/S, uno por cada tipo de dispositivo, mas otro proceso que no corresponde a ningún dispositivo y es la tarea del sistema. Cada proceso de esta capa es

independiente del otro y se comunican mediante mensajes. La primera y segunda capa forman el kernel.

La tercera capa contiene procesos que ofrecen servicios útiles a los procesos de los usuarios. El administrador de memoria (MM) realiza todas las llamadas al sistema de Minix que intervienen en el manejo de la memoria (EXEC, FORK, etc.). El sistema de archivos (FS) efectúa todas las llamadas al sistema de archivos (MOUNT, READ, etc.).

La cuarta capa contiene el shell, editores y los programas del usuario.

Los procesos en Minix siguen el modelo general de procesos. Además puede crear subprocesos que pueden dar origen a más subprocesos creando un árbol de procesos. Las dos llamadas principales para el manejo de procesos son FORK y EXEC. La única manera de crear un nuevo proceso es invocando a la función FORK. Para permitir a un proceso ejecutar otro proceso la función EXEC. Cuando se ejecuta un proceso se la asigna una porción de memoria que está especificada en el encabezado del proceso. Este mantiene esta distribución de la memoria en toda su ejecución, aunque la distribución entre el segmento de datos, segmento de stack y no usado pueden variar durante la ejecución del proceso. Toda la información referente al proceso se guarda en la tabla de procesos que se divide entre el kernel, el administrador de la memoria y el sistema de archivos, donde cada uno tiene aquellos campos que se necesitan. Cuando un proceso comienza a existir o finaliza, el administrador de memoria actualiza su parte de la tabla de procesos y después envía un mensaje al sistema de archivos y al kernel para que actualicen sus partes.

La comunicación entre los procesos se realiza intercambiando mensajes. Cada proceso puede enviar o recibir mensajes de procesos que están en su capa o en la capa inmediata inferior a él. El protocolo de envío y recepción de mensajes es el de rendezvous, o sea que un proceso que envía un mensaje se bloqueará hasta que el receptor lo reciba. Aunque este protocolo es menos flexible, es adecuado para este sistema y mucho más simple ya que no necesita manejo de buffers.

La planificación en Minix es un sistema de lista de espera de múltiples niveles, con tres niveles que corresponden a las capas 2, 3 y 4 de la figura 2. Dentro de cada nivel se utiliza round robin, las tareas del sistema tiene la prioridad más alta, luego vienen el administrador de memoria y el sistema de archivos y por último los procesos del usuario. Cuando se debe elegir un proceso para ejecutar el planificador verifica si hay alguna tarea lista de la cola de mayor prioridad, si hay varias tareas listas

se ejecuta la que está a la cabeza de la cola de espera, si no hay tareas del sistema para ejecutar, pasa al siguiente nivel y trata de elegir al administrador de memoria o al sistema de archivos y si no puede elegir a ninguno entonces pasa a elegir de la cola de los procesos de usuario, y si no puede elegir a nadie pasa a estado ociosos hasta la siguiente interrupción de reloj. En cada interrupción del reloj se verifica si el proceso que se esta ejecutando es una tarea de usuario y si ya ejecutó mas de 100ms (quantum), si se verifica esto el planificador desaloja la tarea y revisa las colas de espera para asignarle el procesador a otra tarea, el proceso desalojado pasa al final de su cola de espera. En resumen, el algoritmo de planificación tiene tres listas de espera con prioridad, una para tareas de E/S, una para los procesos servidores y una para los procesos de usuario. El proceso que se ejecutara es el primer proceso contenido en la lista de espera con la más alta prioridad. Si un proceso de usuario utiliza toda la cantidad de tiempo asignada, se coloca al final de su lista de espera, con lo cual la planificación de los procesos de usuario es de Round Robin con desalojo[TAN88].

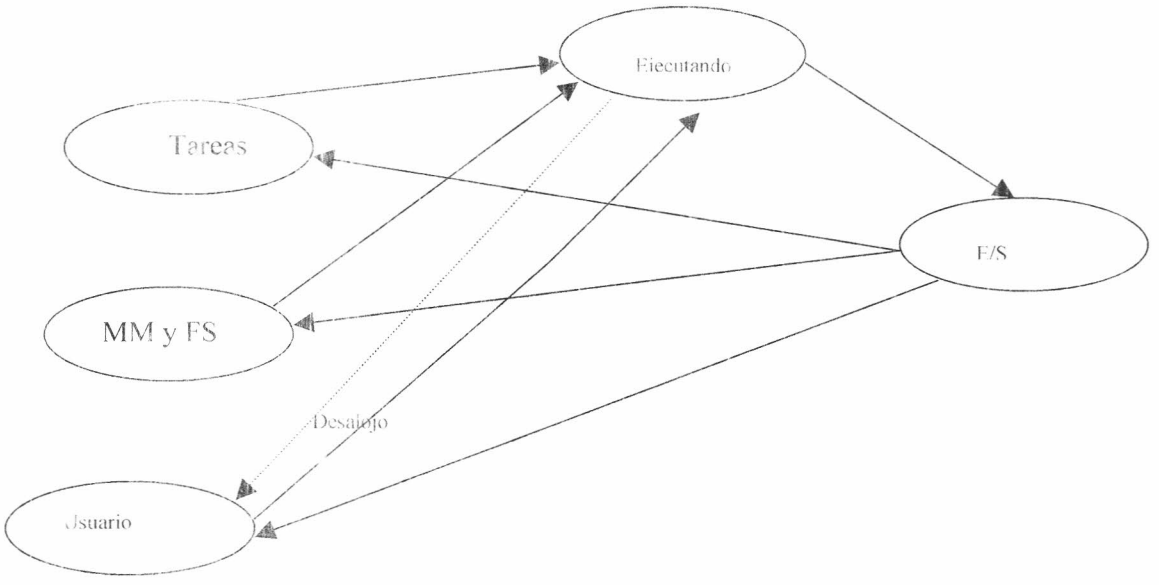


Figura 3 – Diagrama de Transición de Estados.

Las señales se generan de dos formas: por la llamada al sistema KILL o por el kernel. Independientemente de cómo haya sido originada la señal, el administrador de memoria procesa todas las señales de la misma forma. Se realizan chequeos a los procesos que emiten las señales para ver si éstas son válidas. Si está todo bien, se le envía un mensaje a la tarea del sistema a través de kernel para que ponga cuatro palabras (palabra de control, registro CS, contador del programa y

número de señal) en la pila del proceso que emitió la señal. Luego se salvan los registros a la pila y se busca en una tabla la función a ejecutar. Una vez que se ejecutó la función, se restauran los registros y se reanuda el proceso.

5.2. Modificaciones a Minix

En el presente trabajo se propone la construcción de una herramienta para poder medir la perfomance de distintos algoritmos de planificación del procesador para tiempo real. Para ello se necesita un sistema operativo que sea capaz de ejecutar tareas de tiempo real, al que se le pueda cambiar la planificación del procesador. Y además debe proveer las herramientas necesarias para que la herramienta a construir pueda comparar los algoritmos implementados.

Como punto de partida se utiliza el trabajo [WAI97] en el que se presenta un proyecto para proveer facilidades de programación a Minix que pueden ser usadas para el desarrollo de sistemas de tiempo real. Se propone la extensión de los servicios provistos por Minix y se modifica su algoritmo de planificación de tareas. Los servicios extendidos se podrán usar desde un lenguaje de programación ampliando las facilidades existentes para concurrencia, introduciendo facilidades orientadas al cumplimiento de restricciones de tiempo de los procesos en ejecución. El objetivo de este proyecto es el diseño e implementación de algoritmos de planificación que permitan ejecutar tareas de tiempo real dinámicamente, asegurando la planificabilidad de las tareas y obteniendo el menor grado posible de sobrecarga del procesador. El trabajo se basa en el modelo de tareas basado en Tasa Monotónica, que permite determinar si se puede ejecutar un conjunto de tareas en forma predecible. Si el grado de utilización del procesador está por debajo de un límite teórico dado, se garantiza el cumplimiento de las restricciones de tiempo del conjunto de tareas. Por este motivo el implementador no necesita saber en qué momento se está ejecutando cada tarea, con lo que facilita el proceso de desarrollo y las pruebas de correctitud. El modelo de tarea que se ejecuta, ejecuta instancias de una tarea (la mínima unidad de ejecución compuesta por una ráfaga de procesador y una de entrada/salida de dicha tarea), como se muestra en el siguiente gráfico.

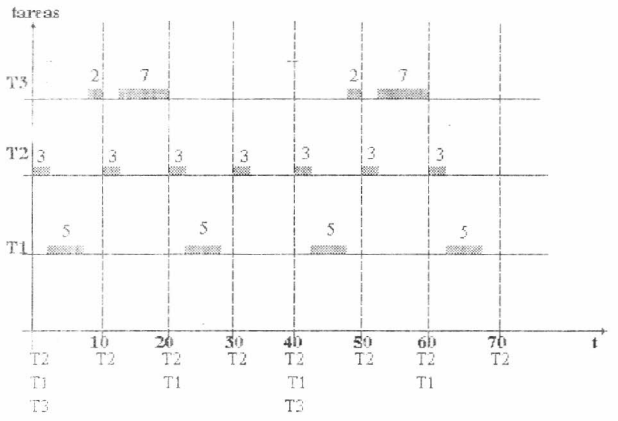


Figura 4 – Diagrama de tiempos

Toda planificación del procesador se iniciará por la interrupción del reloj, y será ella quien decida qué instancias se activan en dicho período. Se desea que el kernel modificado pueda planificar instancias de tareas, diferenciar tareas periódicas de las aperiódicas y las que no sean de tiempo real, lograr alto time-loading y asegurar predictibilidad usando Tasa Monotónica. Con estas consideraciones se realizó un conjunto de modificaciones sobre el código de Minix.

Como las tareas se activarán por la interrupción del reloj, el período de dichas tareas deberá ser múltiplo de la longitud de un tick. La tasa de activación de la rutina del reloj en Minix es de 50 ticks por segundo, por lo tanto el menor período posible para una tarea será de 20ms.

Para mejorar la perfomance del sistema se agregó un conjunto de llamadas al sistema que permiten cambiar el período de activación (granularidad). Aunque dicho cambio provoca mayor sobrecarga, el sistema no alteró su comportamiento con tasas de hasta 5000 tick (con tasa mayores se degrada y con una tasa de 40000 tick queda inoperable).

Se modificó la operación de la rutina de planificación para poder implementar el nuevo modelo de tareas. Se implementó un algoritmo multicolas con remoción, agregando dos nuevas colas de usuario. La cola de mayor nivel administra instancias de tareas de tiempo real y se mantiene ordenada por prioridades usando el algoritmo de Tasa Monotónica. La siguiente cola se administra por Round-Robin (quantum 100ms) y se utiliza para las tareas que no son de tiempo real.

Además de la cola de bloqueados, se agregó al kernel una cola de instancias bloqueadas, administradas por el driver de reloj. Al comenzar un nuevo periodo se remueve la instancia de dicha

cola y se agrega en la cola de tiempo real según su prioridad y si es tiempo de que se ejecute. Al terminar de ejecutar una instancia de la tarea, se pasará nuevamente a la cola de tareas periódicas bloqueadas, hasta que se cumpla un nuevo periodo.

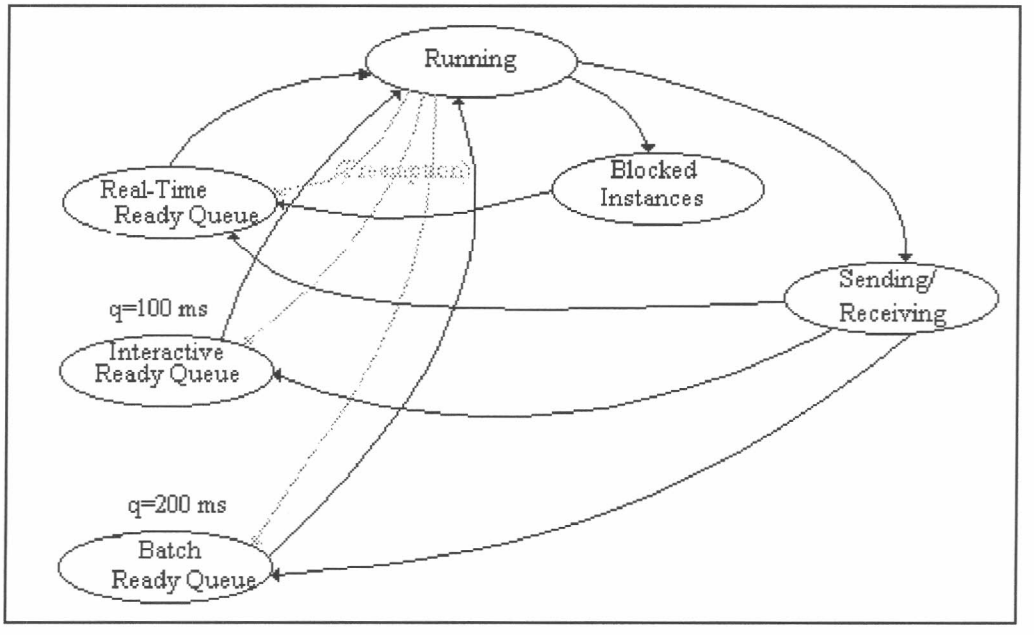


Figura 5 – Diagrama de Transición de Estados modificado

Se puso especial cuidado en la forma de seleccionar las instancias que se ejecutarán en cada intervalo, ya que en cada interrupción del timer deben chequearse todas las tareas, la sobrecarga de procesamiento puede resultar significativa. Se agregó una tabla de activación de tareas para evitar dicha sobrecarga, que contiene un mapa de bits de cada tarea indicando las interrupciones en las que debe ser activada.

También se incorporaron facilidades para que un programador pueda definir tareas con restricciones de tiempo en forma de biblioteca para el compilador C. Se incluyeron primitivas para definir las restricciones de tiempo de las tareas, definiendo el período de una tarea (T_i) y su peor tiempo de ejecución (C_i). La ejecución de esta primitiva provoca que la tarea sea tratada como una tarea de tiempo real. También se incluyó una primitiva para indicar el fin de una instancia. Y otra para indicar que una tarea es aperiódica (indicando también su tiempo de activación y peor tiempo de ejecución).

Con la estructura mencionada anteriormente, se puede asegurar la predictibilidad del conjunto de tareas en ejecución. Aunque la implementación en tiempo de ejecución de los teoremas utilizados para garantizar un conjunto de tareas puede ser costoso en tiempo.

Para garantizar un alto time-loading se usó el algoritmo multicolas, también se permitió el lanzamiento de nuevas tareas en tiempo de ejecución, previo análisis de planificabilidad. La implementación que se propone es más dinámica y con mayor grado de time-loading que la que utiliza el algoritmo de Tasa Monotónica para determinar las prioridades off-line y luego correr las tareas con las prioridades asignadas.

El trabajo propuesto anteriormente en el que se presenta un proyecto para proveer facilidades de programación en Minix que pueden ser usadas para el desarrollo de sistemas de tiempo real, reúne las siguientes características que serán de utilidad para la concreción de los objetivos del presente trabajo:

- Se presenta un algoritmo de planificación modificado para soportar planificación del procesador por Tasa Monotónica y Deadline Driven
- Se provee de tres llamadas al sistema para poder soportar tareas periódicas y aperiódicas: una llamada que indica que una tarea es periódica, otra para indicar que es aperiódica y una última para indicar que termina una instancia de una tarea periódica (*setper*, *setaper*, *inst_end* respectivamente).
- Se provee una llamada al sistema para cambiar la granularidad con la que trabaja la rutina del reloj (*set_grain*).
- Se presenta el sistema operativo preparado para recolectar los datos de las metas perdidas, estos se recolectan en forma separada, las metas periódicas perdidas por un lado y las metas aperiódicas perdidas por otro.
- Se provee una tecla de función por la cual se permite cambiar el algoritmo que se utiliza en la planificación del procesador (cambia de Tasa Monotónica a Deadline Driven).
- Se provee de una tecla de función que muestra los resultados de las metas perdidas por pantalla.
- Se provee una tecla de función que resetea los valores de las variables correspondientes a las estadísticas obtenidas (metas periódicas y aperiódicas perdidas).

En el presente trabajo se pretende construir una herramienta de benchmark, al estilo Hartstone, que sea capaz de comparar distintos algoritmos de planificación del procesador, con una carga sintética

definida por el usuario, y un conjunto de tareas periódicas y aperiódicas también definidas por el usuario. Dicha herramienta debe ser capaz de entregar estadísticas indicando cómo funcionaron los algoritmos de planificación. Para ello se necesita un sistema operativo que responda a las exigencias de esta herramienta. La versión de Minix 1.5 [WAI97] es un buen comienzo, pero le faltan funcionalidades para que esta herramienta pueda ejecutarse.

Las exigencias funcionales que se plantean para el presente trabajo requieren de un sistema operativo capaz de proveer las siguientes funcionalidades:

- Se deberá proveer una facilidad para que un programa sea capaz de cambiar el algoritmo con el que se planifica el procesador sin la intervención del usuario.
- Se deben proveer estadísticas para detectar cómo funciona un algoritmo de planificación dado. Estas estadísticas deben ser las siguientes: cantidad de metas perdidas (periódicas y aperiódicas), tiempo ocioso del procesador, porcentaje de garantía que ofrece el algoritmo con respecto al conjunto de tareas en ejecución.
- Se deben proveer las herramientas necesarias para que un programa sea capaz de obtener las estadísticas calculadas por el sistema.
- Se debe proveer al sistema operativo de las herramientas necesarias para que pueda calcular el límite teórico que garantiza la planificabilidad del algoritmo que está en curso.
- Se deben proveer también las herramientas necesarias para poder calcular el porcentaje de uso del procesador del conjunto de tareas que se está ejecutando en un determinado momento.
- Se le debe proveer al sistema operativo la capacidad de detectar y avisar cuándo se sobrepasó el límite teórico del algoritmo que se está planificando.
- El sistema operativo debe ser capaz de detectar y avisar cuándo se pierde una meta, tanto sea periódica como aperiódica.
- Se debe proveer de una rutina de administración del procesador que presente varias opciones de algoritmos para su planificación.
- Se debe proveer la capacidad de resetear todas las variables estadísticas usadas por el sistema sin la intervención del operador.

El sistema operativo producto del trabajo presentado en [WAI97] presenta algunas de las funcionalidades mencionadas anteriormente pero no están presentes todas ellas. Se puede detectar la perdida de una meta pero no se puede dar aviso de dicho acontecimiento. Tampoco se puede calcular el límite teórico del algoritmo ni del conjunto de tareas en ejecución y no es capaz de

calcular si se está dentro de dicho límite o no y además no se puede dar aviso a la tarea que causa que se pase dicho límite del evento ocurrido. Tampoco se puede cambiar de algoritmo de planificación sin la intervención del operador, ni se pueden obtener las estadísticas recolectadas. En este sistema operativo no se presentan estadísticas de porcentaje de garantía ni de tiempo ocioso del procesador y no se provee de varios algoritmos de planificación del procesador, solo se presentan dos de ellos.

Por consiguiente se tiene que enriquecer al sistema operativo de con algunas funcionalidades, a saber:

- Se debe proveer una llamada al sistema para cambiar el algoritmo de planificación.
- Se debe agregar funcionalidad para recolectar las estadísticas que no se hayan implementado.
- Se debe proveer de una llamada al sistema para obtener los resultados de las estadísticas recolectadas.
- Se debe proveer de un mecanismo para dar aviso a la tarea si el conjunto de tareas en ejecución traspasó el límite teórico del algoritmo en curso.
- Se debe proveer de facilidades para calcular el límite teórico del algoritmo en curso. Y para calcular el porcentaje de uso del procesador del conjunto de tareas que se están ejecutando.
- Se debe proveer de las facilidades para dar aviso si se perdió una meta.
- Se debe poder calcular cuándo el sistema está por llegar a overload y dar aviso de ello.
- Se deben agregar más algoritmos de planificación (Least Laxity First y algoritmos mixtos).
- Se debe poder resetear todas las variables estadísticas usadas.

A los dos algoritmos de planificación ya implementados se le agrega Least Laxity First y se agregan algoritmos mixtos que son una combinación de Tasa Monotónica y Deadline Driven como se detalla en la siguiente tabla:

Algoritmo	Condición
A	Se comienza con Tasa Monotónica, si no hay overload y si existen tareas aperiódicas en el sistema se usa Deadline Driven
B	Se comienza con Deadline Driven, si hay overload se usa Tasa Monotónica
C	Si hay tareas aperiódicas se usa Deadline Driven, si hay tareas periódicas se usa Tasa Monotónica, y si se perdió una meta se usa Tasa Monotónica
D	Si hay tareas aperiódicas se usa Deadline Driven, si hay tareas periódicas se usa Tasa Monotónica, y si se perdió un número especificado de metas se usa Tasa Monotónica

Figura 6 – Algoritmos mixtos

Además se agregan tres nuevas funcionalidades al sistema que sirven para implementar el polimorfismo de desempeño para las planificaciones de Tasa Monotónica y Deadline Driven. Estas funcionalidades son: detectar cuándo se perdió una meta, detectar cuándo se llegó a overload y detectar cuándo se pasa el límite teórico del algoritmo que se está ejecutando.

Entonces la implementación de Least Laxity First, de los algoritmos mixtos y de las funciones de polimorfismo de desempeño, más Tasa Monotónica y Deadline Driven que ya estaban implementados, dejan la siguiente tabla de algoritmos de planificación (codificada en *algtmos.h* - Apéndice B):

Abrev.	Nro.	Descripción
PTM	0	Tasa Monotónica
PDD	1	Deadline Driven
PAA	2	Algoritmo A
PAB	3	Algoritmo B
PAC	4	Algoritmo C
PAD	5	Algoritmo D
PLL	6	Least Laxity First
OTM	10	Tasa Monotónica con polimorfismo de desempeño con la función de overload
ODD	11	Deadline Driven con polimorfismo de desempeño con la función de overload
OAA	12	Algoritmo A con polimorfismo de desempeño con la función de overload
OAB	13	Algoritmo B con polimorfismo de desempeño con la función de overload
OAC	14	Algoritmo C con polimorfismo de desempeño con la función de overload
OAD	15	Algoritmo D con polimorfismo de desempeño con la función de overload
DTM	20	Tasa Monotónica con polimorfismo de desempeño con la función meta perdida
DDD	21	Deadline Driven con polimorfismo de desempeño con la función meta perdida
DAA	22	Algoritmo A con polimorfismo de desempeño con la función de meta perdida
DAB	23	Algoritmo B con polimorfismo de desempeño con la función de meta perdida
DAC	24	Algoritmo C con polimorfismo de desempeño con la función de meta perdida
DAD	25	Algoritmo D con polimorfismo de desempeño con la función de meta perdida

Figura 7 – Tabla de algoritmos de planificación disponibles

En resumen se debe realizar una serie de cambios sobre este sistema operativo, de manera que se convierta en una buena base para poder correr una herramienta de benchmarks que sea capaz de comparar algoritmos de planificación del procesador.

Durante la implementación de los cambios que se le realizaron al sistema operativo del trabajo de [WAI97], se trató de reusar todas las funcionalidades que este proveía, además de mantener las funcionalidades existentes extendiéndolas a las nuevas. En la implementación de algunos cambios no se pudo reusar el trabajo anteriormente mencionado y se tuvo que hacer reformas substanciales al código.

Otra característica de la implementación de los cambios al sistema operativo Minix es la de las llamadas al sistema que se construyeron para que se realicen las diferentes tareas requeridas, se tuvo que construir una rutina en el Administrador de Memoria que atendiera la llamada al sistema y luego le pidiera al Kernel que realizara la tarea requerida. Esta rutina que se construyó en el Administrador de Memoria no hace nada, simplemente es un puente entre el programa de usuario y el Kernel. Esto es así debido a como está implementada la comunicación entre procesos en Minix. Recordemos que es de una capa a otra y que las capas están implementadas como muestra el siguiente gráfico:

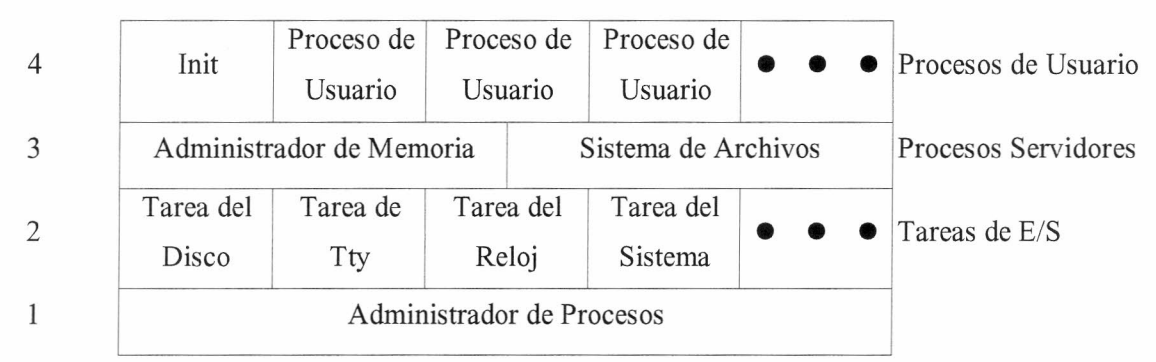


Figura 8 – Estructura en capa de Minix

En este sistema operativo [WAI97] no se provee la facilidad para que se cambie el algoritmo de planificación sin la intervención del operador, se provee de una tecla de función que realiza este trabajo. Los requerimientos funcionales de la herramienta de benchmark exigen cambiar la planificación a pedido de un programa con lo cual se descarta presionar la tecla de función que requiere la intervención del operador.

Se debe implementar un método para que la herramienta de benchmark pueda cambiar de algoritmo de planificación. Para ello se implementó una llamada al sistema. La rutina que atiende esta llamada en el sistema operativo se encargará de setear las variables de ambiente necesarias para que el administrador del procesador use el algoritmo que se pide por programa.

La llamada al sistema que se implementó para cambiar el algoritmo de planificación[1] ingresa un número de algoritmo que esta codificado (ver *algtmos.h* - apéndice B). La codificación que se realizó de los algoritmos de planificación representa todos los algoritmos de planificación que se pueden usar. Se agrega una entrada en esta codificación por cada combinación posible de los algoritmos comunes como Tasa Monotónica, los algoritmos mixtos y el polimorfismo de desempeño.

Se amplía la posibilidad del operador a cambiar manualmente el algoritmo de planificación incluyendo los nuevos algoritmos implementados

La administración del procesador en Minix se resuelve con un sistema multicola de 3 niveles, cada nivel tiene su propio algoritmo de planificación. Las tareas del sistema tienen la prioridad más alta, luego vienen el administrador de memoria y el sistema de archivos, y por último las procesos del usuario. En los dos primeros niveles se utiliza la planificación de fila (FIFO) y en el último nivel se utiliza round robin. Estas colas de listos se implementaron con dos vectores: *rdy_head* y *rdy_tail*. Por cada cola de listos hay una entrada en estos vectores apuntando al principio y al fin de cada lista respectivamente.

Cuando un proceso se desbloquea se coloca al final de su cola de listos. Y si un proceso en ejecución se bloquea o es eliminado por una señal se lo retira de la cola. Por lo tanto en las colas sólo hay procesos ejecutables.

Cuando se debe elegir un proceso para ejecutar, el planificador verifica si hay alguna tarea lista de la cola de mayor prioridad, si hay varias tareas listas se ejecuta la que está a la cabeza de la cola de espera. si no hay tareas del sistema para ejecutar, pasa al siguiente nivel y trata de elegir al administrador de memoria o al sistema de archivos y si no se puede elegir a ninguno entonces elige de la cola de los procesos de usuario, y si no puede elegir a nadie se pasa a estado ocioso hasta la siguiente interrupción de reloj. Esta tarea se realiza en *pick_proc*. Destacamos que el proceso, al ser seleccionado para su ejecución, no se retira de la cola de listos.

Los procedimientos *ready* y *unready* se llaman para insertar un proceso ejecutable en su cola de listos y retirarlo respectivamente. Cualquier cambio en las colas de listos que podrían afectar a la

[1] *Set_alg*

elección para que se ejecute un proceso requiere que luego se llame a *pick_proc* para que seleccione el próximo proceso a ejecutar. También se llama a *pick_proc* cuando un proceso en ejecución se bloquea debido a un *send* o un *receive* ya que éste abandona la CPU y le toca ejecutar un nuevo proceso. Después de cada interrupción se realiza una verificación para decidir si se debe llamar a *pick_proc* para elegir una tarea.

Aunque la mayoría de las decisiones de planificación se toman cuando un proceso se bloquea o desbloquea, existe otra situación en donde también de efectúa la planificación: cuando la tarea del reloj verifica que el proceso en ejecución excedió su Quantum se llama a sched para mover el proceso del principio de la cola de listos al final. El sistema de archivos, el administrador de memoria y las tareas de E/S no se planifican por Round Robin, sino que se espera que trabajen adecuadamente y se bloqueen después de haber finalizado su trabajo.

A las tres colas de listos existentes se agregó la cola listos de tiempo real (RT_Q). Esta incorporación modifica las prioridades de las colas de listos que se redefinen de la siguiente manera:

1. Tareas del Sistema
2. Administrador de memoria y Sistema de Archivos
3. Procesos de Tiempo Real
4. Procesos del usuario.

Además de la cola de listos de tiempo real, se agregó al kernel una cola de instancias bloqueadas, administrada por el driver de reloj como se muestra en la siguiente figura. Al comenzar un nuevo período de cada tarea se remueve el proceso que le toca ejecutar de la cola de instancias bloqueadas y se agrega en la cola de tiempo real según su prioridad. Al terminar de ejecutar una instancia de la tarea, se pasará nuevamente a la cola de instancias bloqueadas, hasta que se cumpla un nuevo período. Este trabajo se realiza en *clock_handler*.

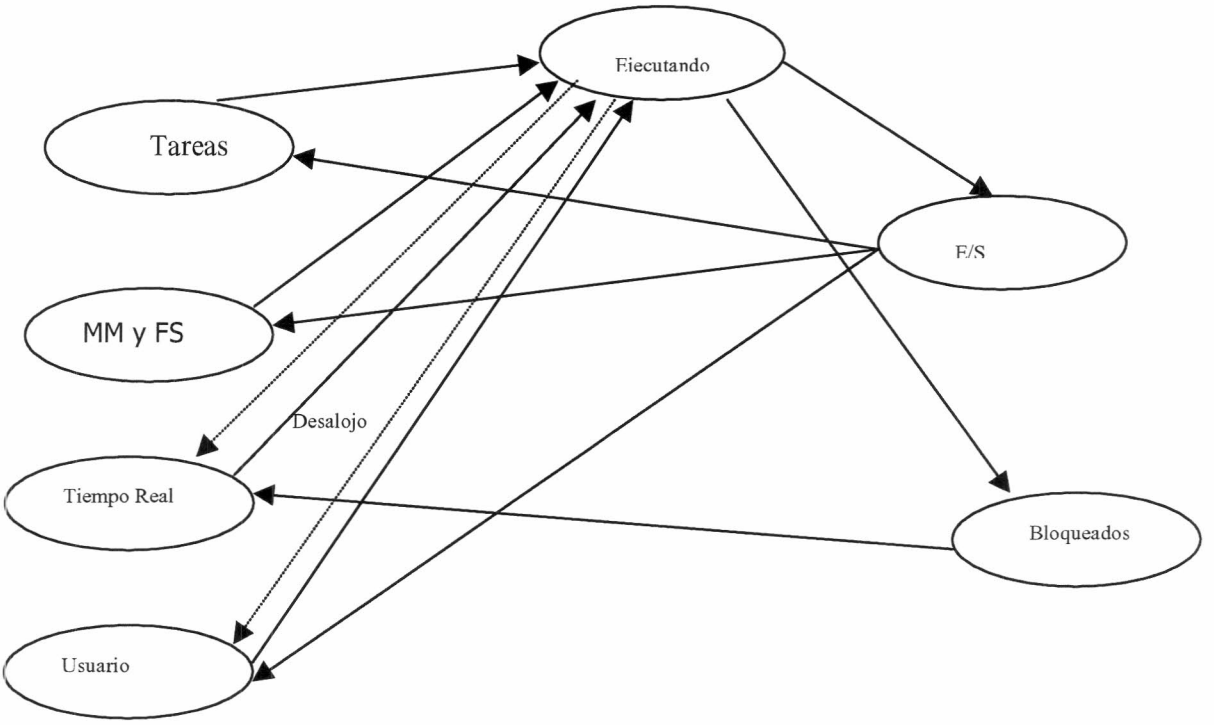


Figura 9 – Diagrama de Transición de Estados del Minix modificado

La rutina *ready* se usa tanto para retirar un proceso de usuario de la cola de bloqueado como para retirar un proceso de tiempo real de la cola de instancias bloqueadas. Si el proceso es de tiempo real se lo inserta[2] en la cola de listos de tiempo real (*RT_Q*) según la planificación que esté en curso. Las planificaciones implementadas son:

- Tasa Monotónica
- Deadline Driven
- Least Laxity First

Notar que las demás planificaciones son una combinación de las dos primeras.

[2] *insertar*

La cola de listos de tiempo real se mantiene ordenada y el criterio depende del algoritmo que se esté ejecutando. Para que la lista se ordene por Tasa Monotónica se usó el periodo de la tarea (T_i). Si se usa la planificación por Deadline Driven se uso como criterio la meta más próxima (*next_period*) y si la planificación es Least Laxity First se uso el tiempo remanente para ejecutar ($T_i - C_i$).

Para llevar a término las modificaciones propuestas anteriormente, se modificó la tabla de procesos agregando funcionalidades para indicar si una tarea es periódica o no, para indicar el grado de criticidad de la misma, para almacenar el T_i y C_i de la tarea, y para indicar el próximo periodo de activación de la misma. Estos cambios se reflejan en la siguiente tabla:

Variable	Descripción
es_aper	Indica si la tarea es periódica o aperiódica
Critic_degree	Indica el grado de criticidad de la tarea. En Tasa Monotónica es T_i, en Deadline Driven es Next_period y en Least Laxity First es T_i - C_i
Ti	Período del proceso
Ci	Peor tiempo de ejecución del proceso
Next_period	Hora de comienzo del siguiente período

Figura 10 – Campos agregados a la tabla de Procesos

Si bien la lista de algoritmos es extensa en lo que se refiere a la administración del procesador solo se implementaron siete algoritmos de planificación: Tasa Monotónica, Deadline Driven, Least Laxity First y los algoritmos mixtos A, B, C y D (descriptos anteriormente). La implementación de las dos primeras planificaciones se reusaron del trabajo de [WAI97].

Como se implementó un nuevo algoritmo de planificación, fue necesario cambiar la rutina que mantiene en orden la cola de listos. El orden de dicha cola se establece de acuerdo al algoritmo que se está ejecutando. Por consiguiente el criterio para ordenar la cola de listos cuando una tarea ingresó al sistema varía de acuerdo al algoritmo en curso, si es Tasa Monotónica se utiliza el T_i de la tarea, si el algoritmo en curso es Deadline Driven se usa *next_period* y si es Least Laxity First se usa el resultado de $T_i - C_i$. La rutina que inserta un proceso en la cola de listos se modificó levemente para que acepte estos tres valores, en vez de los dos valores que aceptaba anteriormente.

Se tuvo que modificar sustancialmente el código del sistema operativo para que éste detectara en qué algoritmo de planificación se encontraba. En este punto no se pudo utilizar la implementación anterior y se agregó código para soportar los nuevos algoritmos de planificación.

Para la implementación de los nuevos algoritmos de planificación, sobre todo para la implementación de los algoritmos mixtos, se necesitó agregar código para calcular el overload del sistema cuando se perdió una meta, el porcentaje de uso del procesador de las tareas de tiempo real que se están ejecutando en el sistema, el tiempo ocioso del procesador y el límite teórico del algoritmo que se está ejecutando.

Para calcular el porcentaje de uso del procesador de las tareas de tiempo real que se están ejecutando en el sistema se utilizo la siguiente fórmula:

$$\sum \frac{C_i}{T_i}$$

Para ello se tuvo que calcular el C_i/T_i (peor tiempo de ejecución/periodo de la tarea) de cada tarea que ingresa al sistema y es de tiempo real. Luego este valor se debe ir acumulando en el caso de que la tarea ingrese al sistema y se debe decrementar si la tarea deja el sistema. El calculo de Ci/Ti se realizó en las rutinas que lanzan una tarea periódica y aperiódica y en la rutina que se ejecuta cuando una tarea deja el sistema (*set_per*, *set_aper* y *do_xit* respectivamente). En las dos primeras se sumó este valor al acumulador y en la última se lo restó. No se realizó ninguna modificación en la rutina que se ejecuta cuando una instancia de una tarea periódica termina (*do_inst_end*) porque en este caso la tarea sigue existiendo en el sistema.

El porcentaje de garantía que presenta un algoritmo para un conjunto de tareas ejecutando responde a la siguiente formula:

$$\sum \frac{C_i}{T_i} \Big/ M$$

Donde M es el total de metas perdidas del algoritmo que se está ejecutando con el conjunto de tareas que está en el sistema.

La implementación de esta medida consiste en acumular la suma de C_i/T_i para cada tarea periódica y aperiódica que se ejecute (para ello se modificaron las rutinas de atención a las llamadas al sistema *set_per, set_aper*). Y en el momento de mostrar u obtener el resultado, se divide por la cantidad de metas periódicas y aperiódicas perdidas. Para que el calculo quedara consistente con el "arreglo" anterior debido a que Minix no trabaja con punto flotante y para que se trabaje con las mismas unidades, se multiplicó a M por 10.000.

Los algoritmos de Tasa Monotónica, Deadline Driven y Least Laxity First poseen una fórmula que si se cumple, garantiza que el conjunto de tareas que se está ejecutando en ese momento no perderá sus metas. Dicha formula para el algoritmo de Tasa Monotónica es:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n.\left(2^{1/n} - 1\right)$$

Y para Least Laxity First y Deadline Driven es:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

La suma de C_i/T_i , necesaria para las fórmulas de estos tres algoritmos se calculó para obtener los valores del porcentaje de garantía y el porcentaje de uso del procesador. Resta calcular la segunda parte de estas ecuaciones. En el caso de Least Laxity First y Deadline Driven no hay problema y el valor anteriormente mencionado se compara con 1. El problema para calcular la segunda parte de la ecuación para la planificación de Tasa Monotónica es que ésta involucra operaciones matemáticas que la librería de Minix no provee, y como no es el objetivo del presente trabajo proveer dichas funciones matemáticas a Minix, y aunque el cálculo de

$$n \cdot \left(2^{1/n} - 1\right)$$

tiende a ln 2 cuando n tiende a infinito (aproximadamente el 69% del uso del procesador); n en las pruebas realizadas no tiende a infinito, por lo tanto esta cota no se puede utilizar.

Cuando el procesador no encuentra ninguna tarea para ejecutar, ni del sistema, ni de tiempo real, ni del usuario, ejecuta la rutina IDLE. En ese momento se dice que el procesador esta ocioso. Se agregó al sistema una acumulador que contabiliza el tiempo en ticks que el procesador gasta ejecutando esta rutina.

Cuando se termina de ejecutar una instancia de una tarea periódica (*do_inst_end*) y cuando una tarea periódica termina, o sea se retira del sistema (*eliminar_per*) los dos procedimientos retiran a la tarea de la cola de instancias bloqueadas y uno la pasa a la cola de listos (*do_inst_end*), el otro procedimiento no porque la tarea se retira del sistema, en ese momento en los dos procedimientos el sistema esta en condiciones de verificar si se perdió una meta entonces se compara el valor del próximo período a ejecutarse y el de la hora actual y si es menor el del próximo período a ejecutarse quiere decir que perdió una meta, la tarea no podrá ejecutarse a término pues perdió su hora de activación. Allí se discrimina si la tarea es periódica o aperiódica y se acumula la falta donde corresponde. También es allí donde el sistema avisa a la tarea en cuestión que se perdió una meta, si corresponde.

Cuando el procesador llega a overload quiere decir que esta al 100% de su uso, para darnos cuenta que ello sucede, se calcula el porcentaje de uso del procesador, cada vez que una tarea de tiempo real ingresa al sistema como se explicó anteriormente (*set_per* y *set_aper*) y se compara con 1 (que es el 100% del uso de procesador)

Con lo implementado anteriormente se está en condiciones de implementar los algoritmos de planificación del procesador propiamente dichos.

La implementación de los algoritmos se distribuye entre las rutinas de atención a las llamadas que indican que una tarea es periódica o aperiódica, la rutina que se ejecuta cuando una tarea termina, la rutina que se ejecuta cuando termina la instancia de una tarea periódica y la rutina que ordena la cola de listos.

No fue necesario modificar el código del sistema operativo para la implementación de los algoritmos de Tasa Monotónica y Deadline Driven debido a que esta implementación está presente en [WAI97]. Sin embargo y como se mencionó anteriormente se modificó levemente el código de la rutina que ordena la cola de listos.

El algoritmo A comienza ejecutando con Tasa Monotónica, y si llegan tareas aperiódicas al sistema se pasa a Deadline Driven siempre que el sistema no esté en overload. Para la implementación de este algoritmo se modificaron las rutinas que atienden la llamada al sistema para indicar que una tarea es periódica, la que indica si es aperiódica, la rutina que se ejecuta cuando una tarea termina y la rutina que atiende la llamada para el cambio de algoritmo (*do_periodic*, *do_aperiodic*, *do_xit* y *do_set_alg* respectivamente). En las tres primeras rutinas mencionadas se utilizó el código ya implementado que calcula el porcentaje de uso del procesador y el que calcula el overload del sistema. Se incorporó código en *do_periodic* para cambiar a la planificación a Tasa Monotónica si hay overload. En la rutina *do_aperiodic* se incorporó código para que cambie a Tasa Monotónica si hay overload y si no que cambie a Deadline Driven, ya que se están incorporando tareas aperiódicas. Y finalmente en la rutina *do_xit* se incorporó código para verificar si se debe pasar a Tasa Monotónica porque hay overload o a Deadline Driven porque quedan tareas aperiódicas ejecutando en el sistema. En la rutina *do_set_alg* se inicializa las variables del sistema para que se comience con la planificación de Tasa Monotónica.

El algoritmo B comienza ejecutando en la planificación de Deadline Driven y si hay overload pasa a Tasa Monotónica. Para la implementación de este algoritmo se tuvieron que modificar las rutinas que atiende la llamada al sistema que indica que una tarea es periódica, la que indica que es aperiódica, la rutina que se ejecuta cuando una tarea termina y la rutina que atiende la llamada para el cambio de algoritmo (*do_periodic*, *do_aperiodic*, *do_xit* y *do_set_alg* respectivamente). Aquí también se reusa el código que calcula el overload del sistema y el porcentaje de uso del procesador. En las rutinas *do_periodic*, *do_aperiodic* y *do_xit* se incorporó código para cambiar a la planificación a Tasa Monotónica si hay overload. En la rutina *do_set_alg* se inicializan las variables del sistema para que la planificación comience por Deadline Driven.

El algoritmo C comienza ejecutando con Tasa Monotónica, si se ingresan al sistema tareas aperiódicas cambia a la planificación de Deadline Driven y si se pierde una meta cambia nuevamente a Tasa Monotónica. Para la implementación de este algoritmo se tuvieron que modificar las rutinas que atiende la llamada al sistema para indicar que una tarea es aperiódica, la rutina que se ejecuta cuando una tarea termina, la rutina que atiende la llamada para el cambio de algoritmo y la rutina que se ejecuta cuando se termina de ejecutar la instancia de una tarea periódica (notar que termina la instancia de la tarea y no la tarea) (*do_aperiodic*, *do_xit*, *do_set_alg* y *do_inst_end* respectivamente). En *do_aperiodic* se incorporó código para que la planificación pase a Deadline Driven ya que se están ingresando tareas aperiódicas, en *do_set_alg* se inicializan las

variables del sistema para que la planificación comience con Tasa Monotónica. En *do_inst_end* y *do_xit* se verifica que la tarea que abandona la cola de instancias bloqueadas no haya perdido su meta, si es así se pasa la planificación a Tasa Monotónica; y en *eliminar_per* se verifica que queden tareas periódicas ejecutando cuando termina una tarea, si quedan tareas periódicas ejecutando se pasa al algoritmo a Deadline Driven si no a Tasa Monotónica.

El algoritmo D comienza ejecutando con Tasa Monotónica, si se ingresan al sistema tareas aperiódicas cambia a la planificación de Deadline Driven y si se pierde un número de metas determinado por el usuario, se cambia nuevamente a Tasa Monotónica. En la implementación de este algoritmo se reusa el código confeccionado para la implementación del algoritmo C, con la salvedad que en la rutina *do_set_alg* se parametrisó con una variable la cantidad de metas periódicas perdida. En el algoritmo C se inicializa el valor de esta variable en 1 y en el algoritmo D la cantidad de metas pedidas se inicializa con lo que determina el usuario. Entonces en las rutinas *do_inst_end* y *do_xit* no se pregunta si se perdió una meta sino si la cantidad de metas perdidas es mayor que esta variable.

Para la implementación del algoritmo Least Laxity First se tuvieron que modificar la rutina que inserta los procesos ordenados en la cola de listos y la rutina que atiende la llamada al sistema cuando se elige un algoritmo (insertar y *do_set_alg* respectivamente). En *do_set_alg* se inicializan las variables del sistema para que la planificación comience con Least Laxity First y en insertar se verifica si la planificación en curso es Least Laxity First ya que se usa como criterio para ordenar la cola de listos : $T_i - C_i$, como se explicó anteriormente.

Se incluyó código en todas las rutinas mencionadas anteriormente para que se descrimine la planificación en curso y se tome acción correspondiente a dicha planificación, de esta manera cada planificación es independiente de la otra.

Para implementar el polimorfismo de desempeño se necesita de un mecanismo por el cual el sistema operativo le avise a la tarea que causa el evento, ya sea que produce overload, pierde una meta o hace pasar al conjunto de tareas del límite teórico, de lo ocurrido, así ella puede tomar acción. Este mecanismo es la implementación de señales del sistema operativo hacia la tarea.

Se debe tener especial cuidado con el uso de las señales en Minix debido a que este sistema operativo mata a una tarea que recibe una señal que no esta esperando. Por ello se dividieron las

planificaciones disponibles para el procesador descriminandolas por función de polimorfismo de desempeño así se asegura que el programador cuando pide que, por ejemplo, la planificación de Tasa Monotónica le avise cuando perdió una meta, éste seguro que va a poder manejar la señal que le llega del sistema operativo.

Otro tema a tener en cuenta es que Minix es capaz de manejar solo 16 señales y solo tiene dos números de señal desocupados, y los requerimientos para ampliar este sistema operativo requieren la implementación de tres señales, con lo que se estuvo ante dos alternativas. O se ampliaba el número de señales de Minix o se usaba alguna señal que se supiera el sistema no use en el momento de las pruebas. Se decidió por la segunda alternativa debido a que fue la más barata porque Minix tiene previsto señales para el uso de modem y estas señales no se usaran durante las pruebas del sistema.

Si se desea que la tarea periódica o aperiódica reciba la señal que le enviará el sistema operativo por alguna de las funciones de polimorfismo de desempeño implementadas, la misma deberá tener las siguientes características:

- Informarle al sistema operativo que manejará[3] la señal por medio de una función y debe informarle que señal es la que manejará.
- Manejar la señal que se lo informa en la misma llamada al sistema del punto anterior.

La función que maneja la llamada al sistema queda a cargo del programador. Esta función puede hacer que se termine la tarea o que el ciclo con el que realizan los cálculos sea menor o lo que el modelo que se está probando requiera. No existen restricciones para la implementación de esta función.

El sistema operativo detecta que una tarea perdió una meta en dos oportunidades como ya dijimos anteriormente: cuando una instancia de una tarea deja de ejecutar[4] y pasa a la cola de instancias bloqueadas de la cola de listos, y cuando la tarea deja el sistema[5] y se elimina de la cola de instancias bloqueadas.

---

[3] *Signal(señal_id, funcion_id)*
[4] *do_inst_end*
[5] *do_xit*

En ambas oportunidades se chequea que la hora actual sea mayor que la hora de activación del próximo periodo. Si esto es cierto entonces se manda un mensaje al proceso indicando que perdió una meta pues se pasó la hora de activación del próximo periodo de la tarea, o sea el presente período no le alcanzó para completar su ejecución. Se implementó el código de tal manera que solamente se le envía el mensaje a la tarea si se está ejecutando Tasa Monotónica o Deadline Driven con la función de polimorfismo de desempeño de perdida de meta. Esto se realiza así debido al problema de Minix en donde una tarea que no esta esperando la señal al recibirla se muere.

Cuando una nueva tarea ingresa al sistema tanto sea ésta periódica o aperiódica se chequea si la misma provoca sobrecarga del procesador. Esto se realiza en las rutinas *set_per* y *set_aper*. Para medir la carga del procesador tomamos la suma acumulada de la relación C_i/T_i y esto lo comparamos contra 1 (que seria el 100% del uso del procesador). Entonces la comparación de la sobrecarga es contra 10.0000. Si se eligió planificación de Tasa Monotónica o Deadline Driven con la función de polimorfismo de desempeño de overload y el cálculo es mayor que 1, entonces el sistema manda una señal a esta tarea para avisarle que causa overload.

Cuando una tarea nueva ingresa al sistema y es de tiempo real, hay que recalcular el porcentaje de uso del procesador incrementando con los valores de esta nueva tarea. Esto se realiza en *set_per* y *set_aper* como se explicó anteriormente. Si se tiene calculado este valor se puede saber si el nuevo conjunto de tareas que se está ejecutando está dentro de la garantía que ofrece el algoritmo en curso. Para ello se compara el nuevo valor con el límite teórico del algoritmo calculado como se explicó anteriormente. Si el nuevo valor es mayor que este límite y si el algoritmo de planificación que se esta ejecutando es Tasa Monotónica o Deadline Driven con la opción de polimorfismo de desempeño de límite teórico, entonces se manda un mensaje a la tarea indicando dicho evento.

Hasta este momento ya se implementaron todas las funcionalidades del sistema operativo y falta que se implementen las herramientas necesarias para poder medir el desempeño de un algoritmo de planificación del procesador.

Para poder comparar los distintos algoritmos se necesita alguna medida de desempeño de los mismos. El desempeño de un algoritmo se puede conocer por la cantidad de metas que pierde, por el uso de procesador, por la estabilidad del algoritmo (como se comporta ante sobrecarga), etc.

De todas las medidas propuestas para medir el desempeño de un algoritmo, el sistema operativo del trabajo de [WAI97] provee solo las siguientes:

- Cantidad de metas periódicas perdidas.
- Cantidad de metas aperiódicas perdidas.

La medición de metas perdidas, tanto periódicas como aperiódicas, da una idea parcial del comportamiento de un algoritmo. Para completar el estudio del comportamiento de los algoritmos se necesitan otras medidas:

- Porcentaje de Garantía
- Tiempo ocioso del procesador
- Suma de C_i/Ti

El Porcentaje de Garantía mide la cantidad de metas que se pierden, ponderada por la criticidad de la tarea. En nuestro caso como se implementaron algoritmos mixtos basados en la combinación de algoritmos, la criticidad de una tarea se tomará de un registro implementado en la tabla de procesos (*critic_degree*). Se deja para un futuro considerar la implementación de una llamada al sistema que le cambie la criticidad a las tareas. En esta implementación la criticidad de las tareas se medirá en los tres planificaciones tradicionales por el T_i.

El tiempo ocioso mide la cantidad de tiempo que el procesador no tiene tareas para ejecutar. Esto nos puede indicar el tipo de carga que tiene el procesador.

La suma de C_i/T_i se usa para detectar la predictibilidad de las tareas que se ejecutan, esta medida ayuda a calcular la cota teórica de un algoritmo.

De todas las medidas mencionadas anteriormente, se implementaron:

- Porcentaje de Garantía
- Tiempo ocioso del procesador
- Suma de C_i/T_i
- Cantidad de metas periódicas perdidas.
- Cantidad de metas aperiódicas perdidas.

Para que un programa pueda acceder a los resultados estadísticos recolectados por el sistema se implementó una llamada al sistema por cada métrica anteriormente mencionada. Se implementó la captura de los datos de esta manera porque la programación de las llamadas al sistema es más simple, permitiendo menor tiempo de respuesta y así tener menor interferencia en las estadísticas.

Así como se necesita obtener los resultados de las estadísticas recolectadas también se necesita resetear el valor de las mismas para poder comenzar una nueva prueba. Se implementó una llamada al sistema para realizar esta tarea, se incluyó código para que cuando se ejecute esta llamada todas las variables estadísticas mencionadas anteriormente se inicialicen en cero. Lo que no se inicializa es el valor de las variables del sistema involucradas en la definición del algoritmo de planificación.

Se tuvo que modificar la llamada al sistema que indica que una tarea es aperiódica porque surgió la necesidad de incluir un tiempo de activación a dichas tareas. Se hizo evidente durante las pruebas de la herramienta de benchmarck y sobre todo en los casos de prueba donde se ejecutaba el algoritmo de Tasa Monotónica y las tareas tenían un plan de ejecución bastante apretado, o sea que el C_i de la tarea era casi tan grande como el T_i, que el tiempo promedio de espera del algoritmo no se respetaba e incluso se duplicaba o cuadruplicaba. Este comportamiento se debe a que la herramienta de benchmark moría de inanición debido al algoritmo de administración de las colas del procesador. Las tareas de tiempo real tienen mayor prioridad que las tareas del usuario. La herramienta de benchmark es una tarea de usuario y las tareas que éste lanza son de tiempo real, si el algoritmo lanza un plan de ejecución de las tareas de tiempo real muy apretado, o sea casi sin tiempo libres entre la ejecución de una instancia de una tarea y la otra, entonces no le toca nunca el procesador a las tareas de usuario, o le toca muy pocas veces, con lo que hace que los procesos de usuario mueran de inanición y por consiguiente que el control del tiempo de la prueba no funcione. Como se puede apreciar en el siguiente gráfico de tiempos:

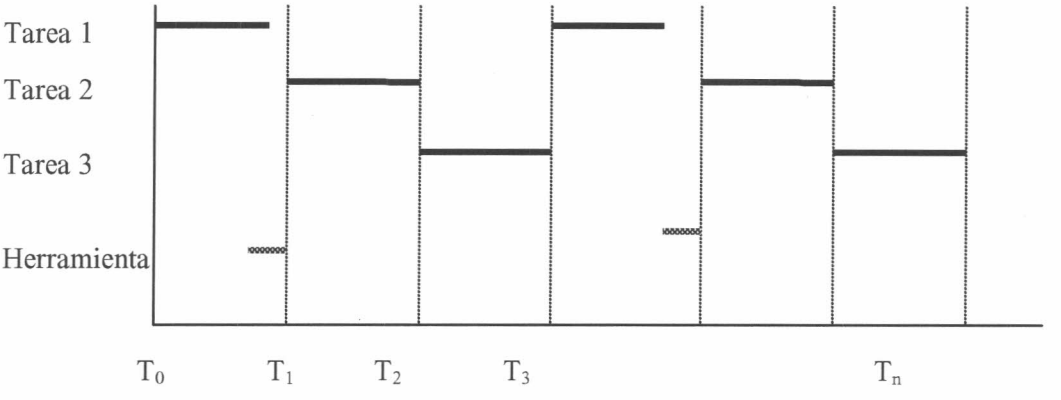


Figura 11- Diagrama de tiempos de la herramienta de benchmark

Para la solución de este problema se tuvieron en cuenta dos alternativas: la primera era subir la prioridad de la herramienta de benchmark a la siguiente, o sea a una prioridad de tiempo real, para que no muriera de inanición y así calculará el tiempo de la prueba correctamente; la segunda alternativa fue que cuando se comience la prueba de la herramienta de benchmark, esta lanzará una tarea aperiódica, única que se despierte a un determinado momento y esta tarea mate a todas las tareas periódicas que existan ejecutando en el sistema.

Cada una de las alternativas tiene sus ventajas y desventajas. La primera alternativa significa que en el momento de la prueba la herramienta de benchmark estará compitiendo con las tareas periódicas y aperiódicas por el procesador. En la segunda alternativa no tenemos competencia por el procesador de una tarea pesada como puede ser esta herramienta, dado que lanzamos una tarea aperiódica que queda en la cola de bloqueados hasta que le toque despertarse. El inconveniente de esta alternativa es que siempre se tiene la presencia de una tarea aperiódica en el sistema y por ello los algoritmos mixtos implementados que funcionan de acuerdo a la presencia de tareas periódicas o aperiódicas en el sistema no funcionaran correctamente dado que siempre tendremos una tarea aperiódica en el sistema.

Después de evaluar las dos alternativas se decidió por la segunda, dado que el sistema seguirá funcionando como lo implementamos y las tareas periódicas y aperiódicas no tendrán competencia de otra tarea que no lo fuera. Preferimos garantizar que el plan de ejecución se realizará tal cuál lo planeado a probar los algoritmos mixtos.

Como resultado de la elección anterior se amplió la definición de una tarea aperiódica. Recordemos que se definía a una tarea aperiódica con el C_i y el T_i de la misma, se agregó a la definición de una tarea aperiódica el tiempo de activación, indicando que una vez que se ingresa la tarea al sistema, esta debe esperar n tick de reloj para comenzar. Se amplió la llamada al sistema que indica que una tarea es aperiódica con un tercer parámetro A_i (tiempo activación de la tarea en ticks). También se tuvo que modificar la rutina de atención de esta llamada, dado que cuando se ingresaba una tarea aperiódica al sistema, esta se colocaba en la cola de listos y ahora si se ingresa una tarea aperiódica al sistema con tiempo de activación distinto de cero no se ingresa a la cola de listos sino que se ingresa a la cola de instancias bloqueadas compartiendo dicha cola con las tareas periódicas hasta que se cumplan A_i ticks desde su ingreso. Para no modificar la tarea del reloj para que verifique esta situación, se ingresa la tarea aperiódica a la cola de instancias bloqueadas con tiempo de activación igual a la hora actual mas el tiempo de activación (A_i).

Se generaron modificaciones sustanciales al código para poder reflejar este cambio: se modificó la llamada al sistema para indicar que una tarea es aperiódica agregándole un parámetro más para indicar el tiempo de activación. Se tuvo que cambiar el tipo de mensaje que se utilizaba para avisar al sistema operativo debido a que en el tipo de mensaje original no se proveía de un la suficiente cantidad de parámetros para pasar todos los valores. En la rutina que atiende a esta llamada en el sistema operativo (*do_aperiodic*) se tuvo que hacer cambios radicales. Si el tiempo de activación pasado como parámetro es 0 (cero) se dejo la rutina como estaba y si no, se agregó código para que dicha tarea aperiódica se agregue a la cola de instancias bloqueadas y se inicializo el campo *next_period* (indica cuando se debe activar una tarea, este campo es usado por las tareas periódicas) con el valor de *realtime* + tiempo de activación pasado como parámetro.

Con estos cambios se logró que la tarea aperiódica que tiene tiempo de activación distinto de cero se despierte cuando se le indica.

Otra modificación importante que se realizó al código del presente trabajo y que no estaba planeada fue un "bug" heredado de la versión anterior y como resultado de él, las instancias de las tareas periódicas no ciclaban, una vez concluida la primera ejecución de las mismas, estas terminaban. Este problema se debe a que el puntero a la instrucción en curso se estaba guardando cuando la tarea realizaba la instrucción *do_inst_end*. Y cuando se le asignaba el procesador nuevamente esa tarea seguía desde donde había dejado, o sea que ejecutaba la ultima parte de la tarea y no la parte que debe ciclar (entre el comienzo y la instrucción *do_inst_end*) como se muestra en la figura 12a.

Para lograr que las tareas periódicas se comportaran como tales se tuvo que resetear el puntero al próximo registro cuando se le asigna el procesador a una tarea que estuviera en la cola de instancias bloqueadas (insertar), o sea que la tarea es periódica (figura 12b) .

Ante la imposibilidad de calcular el límite teórico de Tasa Monotónica como lo expresa el teorema y acercalo a la contante 69, y dado que el límite teórico de Deadline Driven y Least Laxity First es 1, que es lo mismo que la cota de overload, se decidió no usar la implementación de el polimorfismo de desempeño para la función de límite teórico.

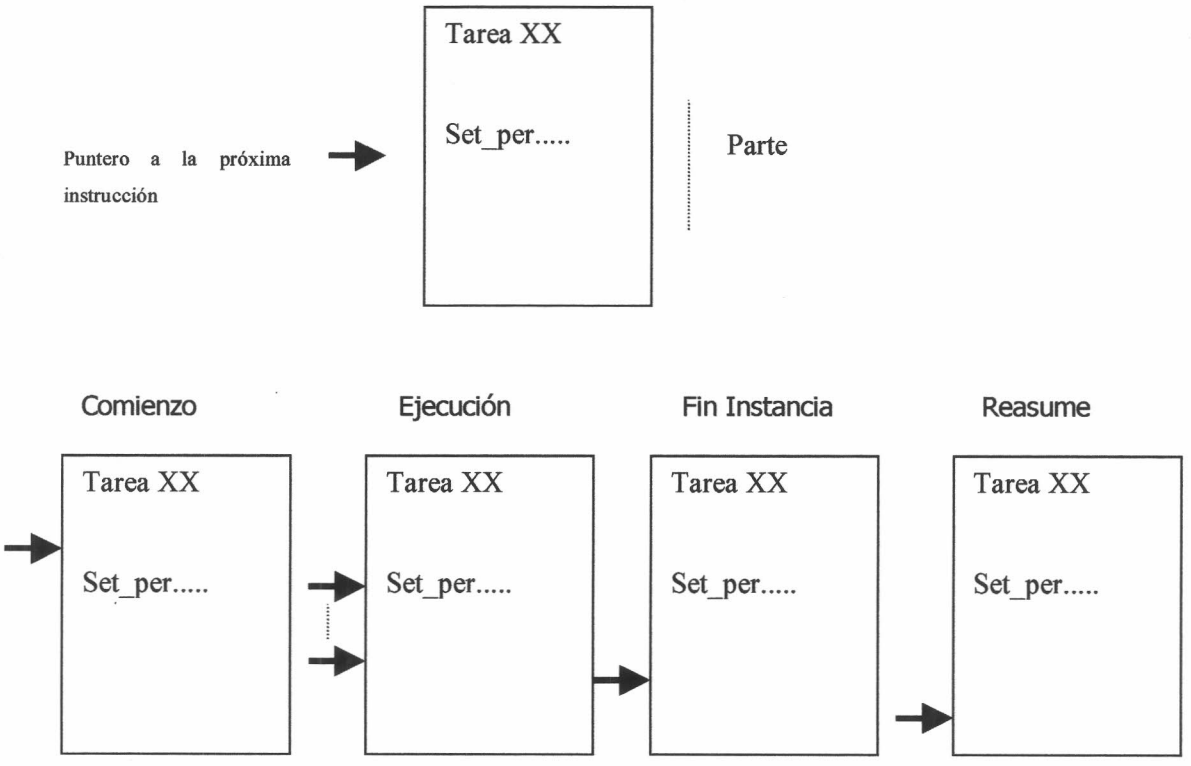


Figura 12 a – Ejecución de una tarea periódica que no cicla

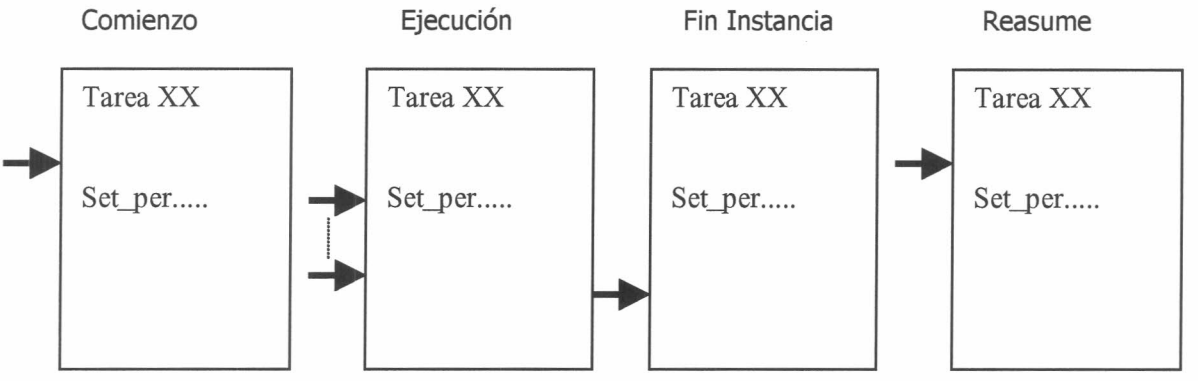


Figura 12 b – Ejecución de una tarea periódica que cicla

Estos cambios afectaron la perfomance del sistema operativo en general de la siguiente manera:

Sistema Operativo	Original	Primera Modificación	Presente Trabajo
Dhrystone	2173	2380	2500

Figura 13 – Performance del sistema.

Estas medidas se tomaron con el benchmark sintético Dhrystone que lo provee Minix. La primera medida corresponde a la medición tomada booteando la máquina con el disco de booteo que provee Minix. O sea que no se ven ninguno de los cambios realizados por este trabajo o por el trabajo de [WAI97]. La segunda medida se tomó bootendo la máquina con un sistema operativo producto de los cambios reflejados de la versión en la que nos basamos, o sea que no se ven los cambios para el presente trabajo. Y la tercera medida se tomó con todos los cambios realizados en el sistema operativo, o sea como esta trabajando en el momento que se tomaron las pruebas de los algoritmos.

Con estas modificaciones el sistema operativo ya está en condiciones de servir como base para la construcción de la herramienta de benchmark.

Como dijimos anteriormente los benchmarks que más se usan y los que son más útiles son los benchmarks sintéticos (en vez de hacer una función útil están construidos sobre la base de la

frecuencia de instrucciones de un conjunto representativo de programas). Los benchmark sintéticos son mejor aceptados que los programas individuales ya que sus resultados se pueden expresar más como una magnitud métrica que en un simple tiempo de ejecución.

En [WEI90] se expresa que sería de gran ayuda para la evaluación de sistemas de tiempo real una familia representativa de requerimientos de benchmarks sintéticos. Y éste es otro de los objetivos del presente trabajo: la construcción de una herramienta para testear el comportamiento de los algoritmos de planificación implementados en las modificaciones al sistema operativo para hacer de éste un sistema operativo de tiempo real.

La herramienta que construimos pretende seguir los lineamientos básicos expresados en [WEI90] además deberá ser capaz de interpretar un archivo de entrada con un conjunto de tareas a ejecutar. Deberá ser capaz de determinar si dichas tareas son periódicas o aperiódicas. Deberá determinar cuando termina una prueba (una prueba no termina cuando se terminan de ejecutar las tareas del conjunto, ya que en el mismo puede haber tareas periódicas y estas no terminan), matar las tareas que se están ejecutando y recolectar datos estadísticos para grabarlos en un archivo de log.

Para la construcción de la herramienta se determinaron los siguientes pasos:

- Detectar las tareas a lanzar desde el archivo de entrada pasado como parámetro al programa. Detectar tanto si la tarea es periódica o aperiódica como el Ci, y Ti de la misma. Para lograr este objetivo se determino que la notación del archivo de entrada tendrá la siguiente forma: [P/A](T_i,C_i). Donde "P" o "A" indican si la tarea es periódica o aperiódica y C_i y T_i , indican el peor tiempo de ejecución de la tarea y el período de su ciclo correspondientes a esa tarea. Cada tarea se separa de la anterior por un ".". O sea que el conjunto de entrada estará representado como sigue

PH:[P/A] (T_i,C_i). [P/A] (T_i,C_i). ... .[P/A] (T_i,C_i);

- donde PH es una indicación para que deje el archivo de salida con determinada extensión, luego se ingresa el conjunto de tareas a planificar separadas por "." y este conjunto termina con ";".
- Realizar un ciclo por cada conjunto de tareas leído que conste de los siguientes pasos:
- Reseteado de las variables estadísticas
- Ejecución de la tarea aperiódica que controla la duración de cada ciclo

- Ejecución del conjunto de tareas a ejecutar (leídas anteriormente). En realidad se ejecuta una tarea periódica y otra aperiódica según corresponda a los parámetros del archivo de entrada cuyos nombres se pasaron como parámetro al programa. Estas tareas pasadas como parámetros deben tener la característica de recibir como parámetro el T_i y el C_i de la tarea.
- Ciclo de espera hasta que la tarea aperiódica lanzada para controlar la ejecución del evento termine.
- Matar las tareas periódicas que se lanzaron.
- Recolectar datos estadísticos.
- Tres eventos de estrés: uno para ver si el sistema es capaz de soportar incrementos cortos de tiempo y cambios de contextos rápidos; otro para testear la capacidad del sistema para manejar un carga incremental pero balanceada y un tercero para testear la habilidad del sistema para manejar grupos de tareas grandes.
- Ejecutar los puntos anteriores para cada algoritmo de planificación que soporte el administrador del procesador.

También se necesito definir un conjunto de tareas auxiliares para el correcto funcionamiento de la herramienta, se definieron las siguientes tareas:

TK: Tarea aperiódica que se despierta después de determinado tiempo. Se usa para determinar la duración de cada ciclo de la herramienta. Para poder implementar esta tarea fue necesario redefinir la llamada al sistema que avisa al mismo que una tarea es aperiódica (*set_aper*)

TP: Tarea periódica de ejemplo y default que se utiliza para que la herramienta ejecute tareas periódicas de carga sintética. La tarea recibe como parámetro su C_i y T_i y ejecuta un ciclo C_i veces ejecutando operaciones matemáticas. Además indica al sistema que utilizará tres funciones para el caso de las señales de overload, límite teórico y perdida de meta, usadas en el polimorfismo de desempeño, también tiene programada estas funciones que determinan que el ciclo en el que se realizan las operaciones matemáticas sea más corto.

TA: Tarea aperiódica de ejemplo y default que se utiliza para que la herramienta ejecute tareas aperiódicas de carga sintética. La tarea recibe como parámetro su C_i y T_i y ejecuta un ciclo C_i veces ejecutando operaciones matemáticas.

El seudo-código de estas tareas se puede ver en el apéndice A. En el diseño de las tareas *TP* y *TA* no se uso el benchmark sintético Drhystone que provee Minix por la sencilla razón que este benchmark es típico de los sistemas comerciales y no refleja la realidad de los sistemas de tiempo real. Dado que este benchmark realiza operaciones con tipos de datos de registros y la característica de los sistemas de tiempo real es que se realizan operaciones matemáticas. Por lo tanto en vez de usar el benchmark Drhystone se decidió proveer la carga sintética de las tareas de tiempo real por medio de operaciones matemáticas. Tampoco se pudo usar el benchmark sintético Whestone más característico de los sistemas de tiempo real porque Minix no lo provee y no es un objetivo del presente trabajo crear este benchmark.

Se definieron los testeos de estrés según [WEI90]. En ese trabajo se propone que cada vez que corra el lote de pruebas se la aplique alguna distorsión para poder ver la reacción del algoritmo testeado en diversas dimensiones. Por lo que se diseñaron los siguientes testeos de estrés:

- Se definió un testeo para que el sistema manejara incrementos cortos de tiempo y que los cambio de contexto se produzcan rápidamente, se implementó este testeo mediante el decremento de 100 al T_i de la tarea con mayor T_i en el conjunto de tareas de entrada.
- Se definió otro testeo para incrementar el trabajo de carga total y así el sistema deberá manejar un trabajo de carga incremental pero balanceado. Esto se logra escalando los T_i de las tareas a ejecutar por 10.
- Se definió otro testeo de estrés para que el sistema maneje grandes grupos de tareas. Esto se consiguió agregando una tarea al conjunto de tareas con el C_i y T_i de la tarea con mayor C_i y T_i.

Las características del conjunto de datos no depende de la herramienta, si no de los datos de entrada del algoritmo, con lo que queda bajo la responsabilidad del programador planear el conjunto de tareas a testear y con esto se logra que se puedan testear los algoritmos bajo distintos puntos de vista (comportamiento con tareas periódicas cortas, largas, con aperiódicas, etc.).

Otra posibilidad que brinda la herramienta propuesta para este trabajo, es que el usuario define la tarea periódica y aperiódica que la herramienta ejecutará. Con lo que se logra gran versatilidad en cuanto a las pruebas que se pueden realizar.

Los requerimientos del benchmark Hartstone sirvieron como base para el diseño de esta herramienta. Entre las características de los requerimientos están las siguientes:

Medir el dominio del problema de tiempo real duro: los requerimientos deben ser representativos del dominio del problema para el cual es diseñado. El usuario será el que defina la carga del test, ya que el usuario puede diseñar el conjunto de tareas lo más parecido posible a su realidad y es él, el que conoce el sistema que está modelando, no sirve tener predefinido un conjunto de tareas a ejecutar, porque ninguno se asemejará a la realidad de los diversos sistemas que se quieren modelar para testear.

Complejidad creciente: Los requerimientos se irán incrementando en complejidad. queda a cargo del programador que determine cargas que vayan incrementando en complejidad.

Testeo de Estrés: Cada test individual será estructurado para tener un requerimiento base y una estrategia para modificar ese requerimiento para forzar al sistema a sus límites a través de un número de dimensiones. Se definieron tres testeos que alteran la carga original de la herramienta de benchmark. Cada uno esta definido para probar una dimensión diferente del sistema: cambios rápidos de contexto, incrementos cortos de tiempo, carga incremental y balanceada, grandes conjuntos de tareas.

Auto - verificación: Cada test individual debe verificar que los cálculos se están realizando correctamente y que se alcanzan las metas. No se pudo respetar en la construcción de esta herramienta este punto. Ya que la herramienta no es capaz de poder modificar la ejecución de una tarea que lanzó para ejecutar ya que no puede decirle a una tarea que no corra porque perdió una meta. No se puede hacer esto por dos razones: una es que el algoritmo una vez que lanza las tareas pierde contacto con ellas hasta que terminan, ya que no está simulando a un sistema operativo multitarea, si no que Minix es un sistema operativo multitarea, no se quiso agregar comunicación entre la tarea y la herramienta de benchmark, para que el usuario no tuviera restricciones al momento de cambiar las tareas a ejecutar; y segundo que si se descarta una tarea, ni bien se pierde una meta, no se podrían probar las opciones de polimorfismo de desempeño por lo menos con la función de perdida de meta.

Trabajo de carga sintético: Una carga sintética garantiza que se esta haciendo una cantidad igual de trabajo en cada sistema corriendo a los benchmark independientes de hardware y software. No se puede garantizar que la herramienta de benchmark cumpla con esta carga sintética debido a que por cuestiones de hacer más versátil a la herramienta, se permite al usuario cambiar la tarea periódica y aperiódica que la misma ejecuta. Junto con la herramienta se provee una tarea periódica y una

aperiódica por defecto que realizan una carga sintética. Dado que se está tratando de probar situaciones de tiempo real, la carga sintética provista para estas tareas son funciones matemáticas y ciclos de ejecución que es lo que se consideró más característico de un sistema de tiempo real, por lo tanto queda en manos del usuario respetar este punto. El poder diseñar las tareas que se ejecutan en el testeo beneficia enormemente al usuario debido a que puede tratar de asemejar las tareas a ejecutar a el tipo de tareas que utilizará en el sistema. Otra ventaja es que puede probar como se comportan los algoritmos si las tareas tienen comunicación entre si.

Figura de mérito relativa: La métrica que indica los resultados de la serie debe ser relativa mejor que absoluta, que distinga claramente la ejecución de la carga de trabajo productiva del overhead del planificador. Para obtener esta figura de mérito relativa, se incorporó en las estadísticas obtenidas el porcentaje de garantía del sistema, que es una medida relativa a la carga del sistema. La cantidad de metas perdidas es una medida en relación al tiempo de ejecución de la prueba, pero dado que el tiempo que se testea cada algoritmo no es un parámetro que el usuario pueda definir, ni depende de la máquina en la que corre, y es el mismo para todos los algoritmos, entonces podemos decir que también es una medida utilizable porque siempre se toma bajo las mismas condiciones.

Dado que la herramienta de benchmark que construimos no es idéntica en sus requerimientos a los requerimientos del benchmark Hartstone, y siguiendo el ejemplo de [WEI90] denominamos a nuestra herramienta de benchmark Hs porque es un sub-conjunto de los requerimientos del Hartstone.

6. RESULTADOS

Del fundamento teórico del presente trabajo se desprende que el algoritmo de Tasa Monotónica es el algoritmo que presenta el mejor plan de ejecución asegurando que se cumplan las metas en un sistema con tareas periódicas e independientes exclusivamente.

El algoritmo Deadline Driven se presenta como la mejor opción para tareas aperiódicas, como contrapartida del algoritmo de Tasa Monotónica. Se tratará de demostrar que en el sistema modificado se cumple esta afirmación.

Se definirá qué característica debe poseer el conjunto de tareas para que los algoritmos de Deadline Driven y Least Laxity First presenten la mejor perfomance.

Se comparará el comportamiento de los algoritmos de Tasa Monotónica, Deadline Driven y Least Laxity First.

Se estudiará el cambio de comportamiento de los algoritmos de Tasa Monotónica y Deadline Driven frente al agregado de polimorfismo de desempeño. Se definirá cual de los dos comportamientos de polimorfismo de desempeño es el mejor.

Se determinará cuál de los algoritmos implementados funciona mejor ante sobrecarga del procesador y cuál se recupera más rápido de la misma.

Se tratará de probar si lo enunciado anteriormente se cumple en el sistema operativo modificado, mediante la prueba exhaustiva de los algoritmos de planificación implementados utilizando la herramienta de benchmark (Hs) construida.

Se analizan los resultados con las siguiente medidas arrojadas por Hs:

- Cantidad de metas periódicas perdidas
- Cantidad de metas aperiódicas perdidas
- Porcentaje de Garantía
- Tiempo Ocioso del procesador

Se dice que un algoritmo es mejor que otro si este pierde menos metas periódicas o aperiódicas que el primero, dado que el objetivo de todo buen algoritmo de planificación para tiempo real es lograr que las tareas cumplan sus metas. Que el algoritmo presente un tiempo ocioso del planificador menor indica que el algoritmo balancea bien su carga y no desperdicia este recurso. A mayor porcentaje de garantía el algoritmo es más estable y se sobrepone mejor ante sobrecargas.

Cabe destacar que si se cambia el algoritmo de planificación durante una prueba no se obtendrán resultados confiables debido a que se estaría ordenando la cola de listos con dos criterios diferentes, habría dos medidas mezcladas hasta que las tareas ordenadas por la planificación anterior abandonen la cola de listos. No es el caso de las pruebas realizadas para el presente trabajo, ya que se tuvo en cuenta esta particularidad y se implementó el Hs de tal manera que solamente cambia de planificación cuando se termina una prueba, o sea que no hay más tareas periódicas ni aperiódicas en la cola de listos.

Para la obtención de los resultados se utiliza el siguiente método de trabajo:

- se definen los lotes de prueba,
- se confecciona un directorio de trabajo para cada lote de prueba numerándolos como se muestra en la tabla del apéndice E.
- en cada directorio de trabajo se crea un archivo que es el archivo de entrada a Hs de cada lote de prueba.
- se confecciona un script que copia cada archivo de entrada al directorio de trabajo del Hs
- se llama a Hs con este archivo de entrada
- se copia las salidas al directorio de trabajo correspondiente a ese lote de prueba.
- se toma el promedios de las distintas corridas para cada algoritmo que deja Hs en el archivo de resultados.

Se define la llamada a Hs de la siguiente forma:

Hs *param1 param2 param3*

Donde *param1* = lote de prueba
param2 = nombre de la tarea periódica
param3 = nombre de la tarea aperiódica

Para los casos cuyo limite teórico es mayor que el limite teórico de Tasa Monotónica, Deadline Driven y Least Laxity First, el algoritmo Tasa Monotónica tiene mejor perfomance que los otros dos. La perfomance de Least Laxity First y Deadline Driven se mantiene igual para ambos algoritmos. En cuanto al polimorfismo de desempeño, el algoritmo de Deadline Driven mejora con la función de pérdida de meta y el de Tasa Monotónica mejora su comportamiento con la función de overload. La función de pérdida de meta tuvo la misma perfomance que Tasa Monotónica.

Para los casos cuyo limite teórico es menor que el limite teórico de los tres algoritmos principales estudiados, Tasa Monotónica nuevamente presenta la mejor perfomance en cuanto a tareas periódicas. La perfomance de Least Laxity First es muy pobre frente a la de los otros dos. Con las tareas aperiódicas es nuevamente Tasa Monotónica quien mejor se comporta. En cuanto a las pruebas realizadas con el polimorfismo de desempeño, en este tipo de casos, el algoritmo de Deadline Driven no mejora y el de Tasa Monotónica mejora su perfomance usando la función que avisa la presencia de overload del sistema.

Ante los lotes de prueba conformados solo por tareas periódicas, se observa que Tasa Monotónica se comporta un poco mejor que Deadline Driven, y la perfomance de Least Laxity First, otra vez queda atrás. El polimorfismo de desempeño no mejora a ninguno de los dos algoritmos (Tasa Monotónica y Deadline Driven). Tasa Monotónica es el que menos tareas aperiódicas pierde para este lote de prueba. Nótese que la única tarea aperiódica que se presenta en este caso es TK. Esto se debe a cómo funciona la herramienta de benchmarks (Hs) que para lograr corridas uniformes en el tiempo, se lanza una tarea aperiódica que mata a las periódicas después de un determinado tiempo.

En los lotes de prueba compuestos solo por tareas aperiódicas, se puede observar que la perfomance del algoritmo Tasa Monotónica es superior a la de Deadline Driven y Least Laxity First y estos dos últimos se comportan de manera similar. No es posible sacar conclusiones sobre el polimorfismo de desempeño debido a que esta técnica se utiliza con tareas periódicas.

En el análisis de los lotes de prueba con tareas mixtas, o sea tareas periódicas y aperiódicas, se muestra que la perfomance del algoritmo Tasa Monotónica en cuanto a la pérdida de metas periódicas supera medianamente a Least Laxity First y ampliamente a Deadline Driven. En el caso de las pérdidas de metas aperiódicas, de nuevo Tasa Monotónica se comporta mejor y el resto de los algoritmos lo hacen en forma similar. En cuanto al polimorfismo de desempeño, la función de overload mejora tanto al algoritmo Tasa Monotónica como al de Deadline Driven.

Para los lotes de prueba en donde el C_i de las tareas está muy cerca del T_i, o sea el C_i es mayor que el 50% que el T_i, se verifica que la perfomance en cuanto a la pérdida de metas periódicas de Least Laxity First es notablemente mejor a la de Tasa Monotónica y Deadline Driven y estos últimos se comportaron en forma similar. Para el caso de la pérdida de metas aperiódicas, el algoritmo que mejor se comportó fue Tasa Monotónica seguido de Least Laxity First. El polimorfismo de desempeño accionado por la función de overload mejora la perfomance del algoritmo Tasa Monotónica; la función de pérdida de meta es la que mejora a Deadline Driven.

Para los lotes de prueba en donde el C_i de las tareas es mucho más chico que el T_i, o sea el C_i es menor que el 50% del T_i, se advierte que la perfomance del algoritmo Tasa Monotónica es mejor que los otros dos, y el polimorfismo de desempeño para la función de overload mejora tanto a Tasa Monotónica como a Deadline Driven.

Resumiendo:

Casos de Prueba	Preformance			Polimorfismo de desempeño (Algoritmo que mejoró)		Observaciones
	TM	Deadline Driven	LLF	Overload	Pérdida de meta	
Límite Teórico del algoritmo > Límite Teórico de TM, DD y LLF	Mejor	Igual a LLF	Igual a DD	TM	DD	
Límite Teórico del algoritmo < Límite Teórico de TM, DD y LLF	Mejor	-	Peor	TM	-	
Sólo Tareas Periódicas	Mejor	-	Peor	-	-	
Sólo Tareas Aperiódicas	Mejor	Igual a LLF	Igual a DD	-	-	La técnica de polimorfismo de desempeño se utiliza solo con tareas periódicas
Tareas periódicas y aperiódicas	Mejor	Peor	-	TM,DD	-	
$C_i > 50\% T_i$ (periódicas)	-	Peor	Mejor	TM	DD	
$C_i > 50\% T_i$ (aperiódicas)	Mejor	Peor	-			
$C_i < 50\% T_i$	Mejor	Igual a LLF	Igual a DD	TM, DD	-	

Figura 14 – Comportamiento de los algoritmos

Las distintas formas de polimorfismo de desempeño estudiadas, tanto sea por aviso de overload o por meta perdida, mejoran la perfomance de Tasa Monotónica y Deadline Driven. Particularmente el polimorfismo por overload en todas las oportunidades presenta mejor perfomance que el de meta perdida.

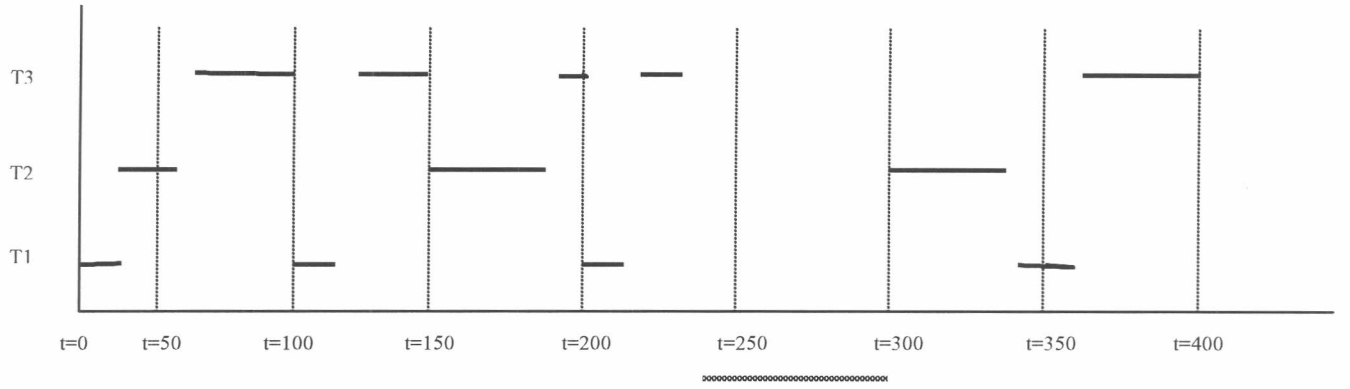
La planificación de todos los algoritmos aprovecha el tiempo del procesador, ya que como se observa en los gráficos del apéndice D[6], la mayoría de los casos no pierden muchas metas y permiten que el procesador tenga poco tiempo ocioso. Se debe tener en cuenta que el uso del procesador se comparte entre el sistema operativo, las tareas de cada lote de prueba y el Hs.

En los casos donde el T_i de las tareas es muy parecido, se ve que se pierde más tiempo debido a que todas las tareas comienzan sus períodos casi al mismo tiempo, por lo tando cuando el procesador termina de ejecutar el lote, se queda esperando hasta el próximo período de las tareas.

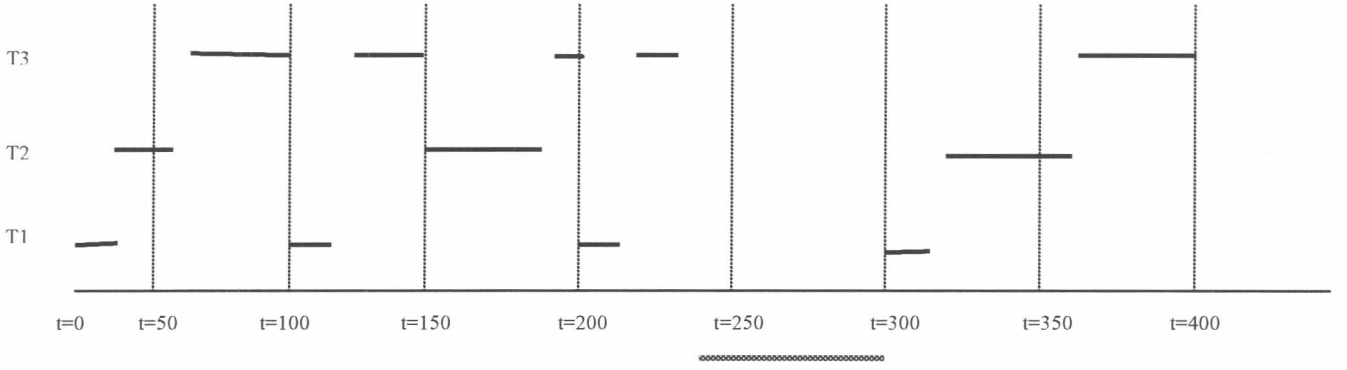
El tiempo ocioso que pierde el procesador esta más relacionado al lote de pruebas que al algoritmo que se ejecuta.

No se logrará la ocupación total del procesador, es decir que el tiempo ocioso recuperado es cero, porque las tareas que se están corriendo en el momento de la prueba son: tareas de tiempo real (se activan a períodos regulares) y el Hs que después de lanzar las tareas realiza un *wait()* quedando en espera de sus hijos. Puede ocurrir que el procesador no tenga nada que ejecutar. Tal es el caso en donde las tareas periódicas están en la cola de instancias bloqueadas esperando su próximo periodo para ejecutar, la tarea aperiódica TK también está en dicha cola porque todavía no le llega la hora de activación, y el HS está bloqueado por el *wait()* que ejecutó para esperar a sus hijos, como se puede observar en el siguiente diagrama de tiempos.

[6] Tiempo Ocioso – Metas Periódicas Perdidas.



Deadline Driven



Tasa Monotónica

Figura15 – Tiempo ocioso del procesador para Tasa Monotónica y Deadline Driven

La estabilidad de los casos testeados es similar en los tres algoritmos principales. En el caso de polimorfismo de desempeño para Deadline Driven la función por overload mejora, pero no mucho la estabilidad de dicho algoritmo. Para el algoritmo de Tasa Monotónica ninguna de las dos funciones de polimorfismo de desempeño mejora la estabilidad del algoritmo original (ver gráficos apéndice D[7]).

A continuación se presenta el análisis del caso de prueba 001 en donde el lote de pruebas se conforma de un conjunto de tareas periódicas e independientes, cuyos valores son los siguientes:

[7] Comparación de la estabilidad de los algoritmos testeados.

Tarea	C_i	T_i
P1	20	100
P2	40	150
P3	100	350

Figura 16 – Lote de prueba el caso 001

El limite teórico de Tasa Monotónica es

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n.\left(2^{1/n} - 1\right)$$

y el de Deadline Driven y Least Laxity First:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Aplicando estas fórmulas a los valores correspondientes al caso se obtiene la siguiente desigualdad:

$$\frac{20}{100} + \frac{40}{150} + \frac{100}{350} = 0.752381 \leq 3\left(2^{1/3} - 1\right) = 0.779763 \leq 1$$

Como se puede ver se verifican los limites teóricos de dichos algoritmos.

De los datos arrojados por Hs para este caso, se puede observar en el siguiente gráfico los resultados obtenidos correspondientes a las metas periódicas perdidas (para las referencias ver apéndice D[8]):

[8] Lotes de Prueba – Cantidad de Metas Periódicas Perdidas

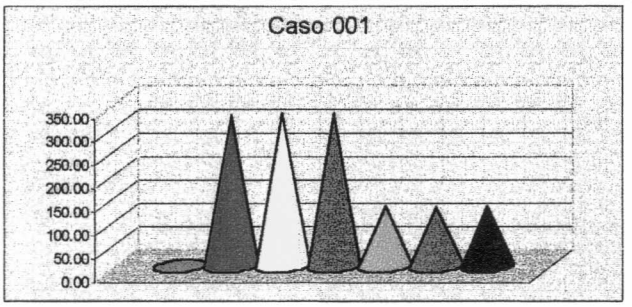


Figura 17 - Cantidad de metas periódicas perdidas[9]

en el que se observa que la planificación de Deadline Driven no pierde metas. Esto es correcto ya que verifica el limite teórico. En los casos de Tasa Monotónica y Least Laxity First se observa que se pierden metas aunque los datos cumplan la garantía teórica porque los resultados que se muestran en el gráfico son un promedio de las corridas de ese lote de prueba para cada algoritmo. Hs corre tres testeos de estrés, como se mencionó anteriormente, modificando el lote de prueba original de la siguiente manera:

Testeo de estrés 1: se decrementa en 100 el T_i de la tarea con T_i mayor, quedando el lote de prueba para este test de la siguiente manera:

Tarea	C_i	T_i
P1	20	100
P2	40	150
P3	100	250

Figura 18 – Lote de prueba del caso 001 después de la primera pasada

Para este primer cambio la desigualdad del límite es:

$$\frac{20}{100}+\frac{40}{150}+\frac{100}{250}=0.86>0.779763$$

[9] Para referencias ir a Apéndice D – Lotes de Prueba - Cantidad de Metas Periódicas Perdidas

Con lo que se observa que para Tasa Monotónica no se cumple la desigualdad del límite teórico, sí para Deadline Driven y Least Laxity First.

Testeo de estrés 2: se decrementa en 10 el T_i de todas las tareas, quedando el lote de prueba:

Tarea	C_i	T_i
P1	20	90
P2	40	140
P3	100	340

Figura 19 – Lote de prueba para el caso 001 después de la primera pasada

Para este cambio la desigualdad del límite queda de la siguiente manera:

$$\frac{20}{90}+\frac{40}{140}+\frac{100}{340}=0.794>0.779763$$

No cumpliendo el límite de Tasa Monotónica nuevamente.

Testeo de estrés 3: se agrega una tarea al conjunto de tareas igual a la tarea de mayor T_i del conjunto. El lote de pruebas se modifica de la siguiente forma:

Tarea	C_i	T_i
P1	20	100
P2	40	150
P3	100	350
P4	100	350

Figura 20 - Lote de prueba para el caso 001 después de la primera pasada

Para este cambio la desigualdad del límite es:

$$\frac{20}{100}+\frac{40}{150}+\frac{100}{350}+\frac{100}{350}=1.02>0.779763$$

Con lo que se observa que no se cumple ningún límite teórico de los tres algoritmos.
Debido a la implementación de los algoritmos en Minix[WAI97], Tasa Monotónica ordena la cola de listos por el T_i de la tarea, Deadline Driven por la meta mas próxima (*next_period*) y Least Laxity First por el $T_i - C_i$ de la tarea. Por esto Tasa Monotónica realiza la mejor elección de la tarea a ejecutarse. Esto sumado a que a través de los testeos de estrés se modifica el Ti de las tareas hace que los algoritmos de Tasa Monotónica y Least Laxity First pierdan metas aunque este último cumpla la garantía teórica.

En el siguiente gráfico muestra la cantidad de metas aperiódicas perdidas para el mismo lote de prueba

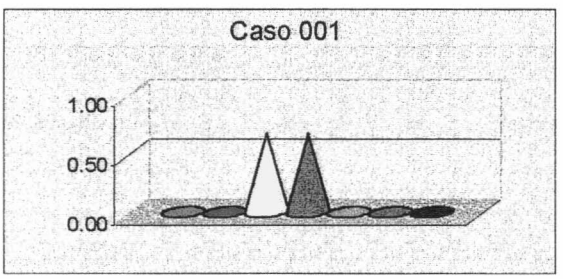


Figura 21 - Cantidad de metas aperiódicas perdidas[10]

Si bien el lote de pruebas no tiene tareas aperiódicas, lo que se observa es que la meta que se pierde es por la tarea aperiódica TK, que no llego a tiempo.

En el siguiente gráfico se observa que las funciones de polimorfismo de desempeño no mejoran el comportamiento de Deadline Driven siendo que en general sí lo hace. Esto se debe a que este caso es un caso típico de tareas que cumplen el límite teórico y que el sistema operativo tiene más trabajo calculando el overload del sistema. Además tiene que enviar a la tarea una señal indicando el overload o la perdida de una meta. A esto se le suma que la carga del sistema esta sobredimensionada por la tarea TK.

[10] Para referencias ir a Apéndice D – Lotes de Prueba - Cantidad de Metas Aperiódicas Perdidas

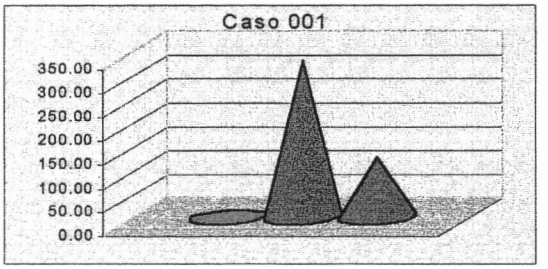


Figura 22 - Cantidad de metas perdidas en Deadline Driven con las funciones de polimorfismo de desempeño[11]

Para el algoritmo Tasa Monotónica se mejora el comportamiento con las funciones de polimorfismo de desempeño. Como se dijo anteriormente, el orden de la cola de listos lo da el T_i de la tarea y al notarse el overload (sobredimensionado) o al detectarse una perdida de meta, la tarea ejecuta menos tiempo liberando antes al procesador y permitiendo que se ejecuten más tareas sin perder su meta.

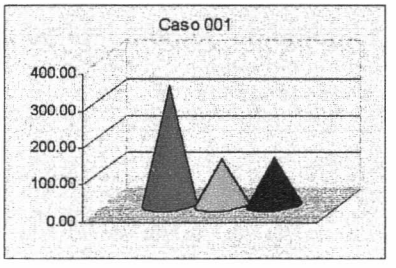


Figura 23 - Cantidad de metas perdidas en Tasa Monotónica con las funciones de polimorfismo de desempeño[12]

En el gráfico de tiempo ocioso se observa que la planificación por Deadline Driven tuvo un buen uso del procesador dado que tiene poco tiempo ocioso. Tasa Monotónica también uso correctamente el procesador, porque aunque haya perdido metas periódicas y aperiódicas no hizo que el procesador estuviera ocioso. Least Laxity First es el que tuvo peor desempeño para este lote de prueba porque no solo planificó mal perdiendo metas, sino que mantuvo al procesador sin nada que ejecutar.

[11] Para referencias ir a Apéndice D – Polimorfismo de Desempeño Deadline Driven
[12] Para referencias ir a Apéndice D – Polimorfismo de Desempeño Tasa Monotónica

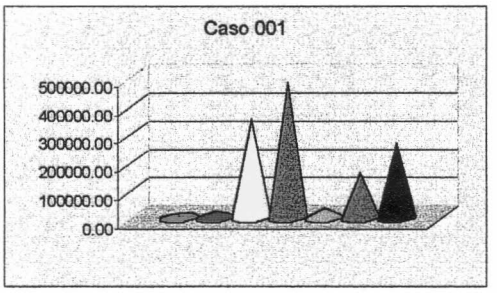


Figura 24 - Cantidad de tiempo ocioso del procesador[13]

Como se muestra en el gráfico, solo Deadline Driven tiene un alto porcentaje de garantía para este caso. Esto se debe a cómo está implementado Deadline Driven en Minix, a los testeos de estrés realizados por la herramienta de medición y por el caso en particular.

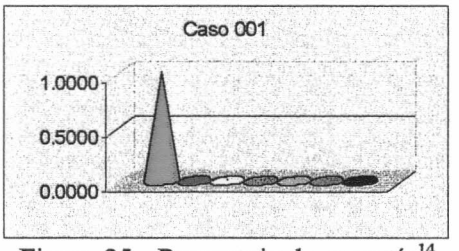


Figura 25 - Porcentaje de garantía[14]

Mientras de ejecutaba el Hs, se observó que el caso 017 "colgaba" al sistema operativo en todas las pruebas realizadas, menos con Deadline Driven. Esto se debe a las características del lote de prueba:

Tarea	Tipo	C_i	Ti
P1	Periódica	1	40
P2	Periódica	1	50
P3	Periódica	1	20
P4	Aperiódica	1	10
P5	Periódica	1	100
P6	Aperiódica	2	40

Figura 26 – Lote de prueba para el caso 017

[13] Para referencias ir a Apéndice D – Lotes de Prueba – Tiempo Ocioso Empleado por el Procesador
[14] Para referencias ir a Apéndice D – Lotes de Prueba – Porcentaje de Garantía

Tarea	Tipo	C_i	Ti
P1	Periódica	40	200
P2	Periódica	80	200
P3	Periódica	160	200

Figura 29 – Lote de prueba para el caso 013

En estos casos de tareas armónicas se observa que el T_i de todas las tareas tanto sean periódicas o aperiódicas, es el mismo (200). Por lo tanto el orden queda establecido por el T_i en el caso de Tasa Monotónica eligiendo las tareas en orden FIFO. En el caso de Deadline Driven como se ordenan por el *next_pediod* y éste se calcula sumando T_i a la hora actual, las tareas también quedan ordenadas por FIFO y en Least Laxity First quedan en orden inverso. Esto hace que cuando se aumenta el T_i de las tareas tanto Tasa Monotónica como Deadline Driven pierdan más metas y en cambio Least Laxity First pierda menos metas.

7. CONCLUSIONES

Los Sistemas de Tiempo Real son aquellos que controlan procesos que ocurren en el mundo real. Los sistemas tradicionales tratan de optimizar factores de acuerdo a la características de la aplicación, ya sea el uso de los recursos, el tiempo de espera de los usuarios, etc. A diferencia de estos sistemas, los sistemas de tiempo real deben poner especial atención a la ocurrencia de "eventos externos" y deben responder a los mismos dentro de un tiempo determinado. Debido a la complejidad del problema, desde el punto de vista de la planificación, las soluciones tradicionales no son útiles para hacer la planificación de tareas, ya que se debe tener en cuenta las metas, los recursos necesarios para una tarea en un tiempo determinado, la precedencia de las mismas y su criticidad.

En este trabajo se analizaron las planificaciones clásicas para sistemas de tiempo real y se le agregaron soluciones de planificación como planificaciones mixtas y polimorfismo de desempeño. Para ello se estudió el comportamiento del sistema modificado en relación al tiempo ocioso del procesador, metas periódicas y aperiódicas perdidas y porcentaje de garantía de los algoritmos.

Luego del análisis podemos concluir que Tasa Monotónica se puede seguir considerando como la planificación adecuada para la carga de trabajo formada por tareas periódicas. También para este algoritmo, el polimorfismo de desempeño por overload lo mejora con cualquier carga de trabajo.

Con respecto al Hartstone, consideramos un método muy útil de testeo para la planificación sistemas de tiempo real ya que permite efectuar chequeos a un conjunto de actividades sobre su conducta determinística, a diferencia de la mayoría de los tests que focalizan su análisis en el tiempo de ejecución de los programas o su funcionalidad.

Finalmente, encontramos de gran utilidad la herramienta Hs, debido a que, al manejar un archivo de texto de entrada, el usuario puede ingresar gran cantidad de lotes de prueba de distintos tipos con lo que queda bajo su responsabilidad planear el conjunto de tareas a testear y con esto se logra que se puedan testear los algoritmos bajo distintos punto de vista (comportamiento con tareas periódicas cortas, largas, con aperiódicas, etc.)

8. BIBLIOGRAFÍA

[ALB93] ALBARRACIN, V.; SCARAMOZZINO, F.; VARELA,S. "Trabajo Práctico N° 4" Laboratorio VIII de Sistemas Operativos. FCEN-UBA. 1993

[BAM89] BAMBERGER, J. et al. "A distributed Ada Real-Time kernel. Version 2". Technical report CMU/SEI-88-TR-17. Software Engineering Institute. Carnegie-Mellon University. April 1989.

[KEN91] KENNY, K., LIN, K. "Building flexible Real-Time Systems using the Flex language". IEEE Computer. May 1991. pp 70-78.

[KLE94] KLEIN, M.; LEHOCZKY, J.; RAJKUMAR, R. "Rate Monotonic Analysis for Real-Time Industrial Computing". IEEE COMPUTER, January 1994. pp. 24-33.

[KLE90] KLEIN, M., RALYA, T. "An analysis of Input/Output Paradigms for Real-Time systems". Technical report CMU/SEI-90-TR-19. Software Engineering Institute. Carnegie- Mellon University.

[LEE] LEE, T.: CHENG, A. "Multiprocessor Scheduling of Hard-Real-Time Periodic Tasks with Task Migration Constraints". Department of Computer Science. University of Houston

[LIU73] LIU, C.; LAYLAND, J. "Scheduling algorithms for multiprogramming in a Hard Real Time System Environment". Journal of the ACM, Vol. 20, No. 1, 1973, pp 46-61.

[SHA91] SHA, L.; KLEIN, M.; GOODENOUGH, J. "Rate Monotonic Analysis for Real- Time Systems". Technical Report CMU/SEI-91-TR-6. Carnegie-Mellon University. Software Engineering Institute. March 1991.

[MIL92] MILLER, F. "The performance of a Mixed Priority Real-Time scheduling algorithm". ACM Operating Systems Review. Vol. 26, No. 4. pp. 5-13.

[MOR95] MORON, C. "A real-time fault-tolerant scheduler". Anais del VI SCTF, Simpósio de computadores tolerantes a falhas. Canela, Brasil. 1995. pp. 313-331.

[RAM94] RAMAMRITHAM, K.; STANKOVIC, J. "Scheduling algorithms and operating systems support for real-time systems". Proceedings of the IEEE, Vol. 82, No. 1, pp. 55- 67, January 1994.

[SPR90] SPRUNT, B., SHA, L. "Implementing sporadic servers in Ada". Technical Report CMU/SEI-90-TR-6. Carnegie-Mellon University. Software Engineering Institute. May 1990.

[STA93] STANKOVIC, J.; SPURI, M. "How to integrate precedence constraints and shared resources in real-time scheduling". CMPSCI Technical report 93-19. University of Massachussets at Amherst. March 1993.

[STA92] STANKOVIC, J. "Real-Time Computing". BYTE, invited paper, pp. 155-160, Agosto 1992.

[STA94] STANKOVIC, J. et al. "Implications of classical scheduling results for Real-Time systems". CMPSCI Technical Report 93-23. University of Massachussets at Amherst. January 1994.

[TAN88] TANNENBAUM, A. "Operating Systems - Design and Implementation". Prentice Hall International Editions, 1987.

[OBE96] Obenza.Ray "The Mathematical Notaton of Theorem 2".Lecture Notes of Rate Monotonic Analisis. Carnegie-Mellon University. Software Engineering Institute.

[WAI93] WAINER, G.; SHANLEY, C. "Informe sobre Sistemas de Tiempo Real y desarrollo de una herramienta para construcción de Sistemas Supervisores de Procesos", en los anales del Encuentro Académico Tecnológico IBM, Resistencia, Chaco. Agosto de 1993.

[WAI94] WAINER, G. "Inclusión de mecanismos de Tiempo Real Duro en un Sistema Operativo de Tiempo Compartido". Anales del II Encuentro Chileno de Computación. 1994. pp. 177-186.

[WAI94c] WAINER, G. "Un curso Teórico/Práctico de Sistemas de Tiempo Real Duro". Anales del III Congreso Iberoamericano de Educación Superior en Computación. 1994. pp. 150-159.

[WAI95] WAINER, G.; BEVILACQUA, R.; ALBARRACIN, V.; NAZAR, M.E. "Estudio e implementación de soluciones alternativas de planificación de procesos en sistemas de tiempo real". Informe interno de Tesis de Licenciatura. FCEN-UBA. 1995.

[WAI95] WAINER, G. "Some Results on Experimental Evaluation of Real-Time Scheduling". En Anales de la XXI Conferencia Latinoamericana de Informática, Panel '95. Canela, Brazil.1995

[WAI95B] WAINER, G. "Algunos resultados de planificación centralizada en Tiempo Real Duro". En Anales de las 24 Jornadas Argentinas de Informática e Investigación Operativa (JAIIO). Buenos Aires, Argentina. 1995.

[WAI95C] WAINER, G. "Implementing Real-Time Services in MINIX". ACM Operating Sistems Review. June 1995.

[WAI97] WAINER, G. "Improving The Performance of Local Real-Time Scheduling". En Proceeding of the 4th IFAC/IFIP Workshop of Algorithms and Architectures of Real-Time Control (AARTC'97). Lisboa, Portugal. 1997.

[WAI97a] WAINER, G. "Sistemas de Tiempo Real. Conceptos y Aplicaciones". FCEN-UBA..Nueva Librería. 1997

[WEI89] WEIDERMAN, N. "Hartstone: synthetic benchmark requirements for Hard Real- Time applications". Technical Report CMU/SEI-89-TR-23. Carnegie Mellon Universty. Software Engineering Institute. 1989

[WEI90] Donohoe, P.; Shapiro, R.; Wiederman, N. "Hartstone Benchmark Results and Analisis".Technical Report CMU/SEI-90-TR-7. Carnegie Mellon University. Software Engineering Institute. 1990

Apéndice A

Pseudo – Código

9. APÉNDICE A : PSEUDO – CÓDIGO

9.1. TP- Tarea Periódica por Default

Función que maneja el evento de Overload
{ Ciclo de 1 hasta 10.000
Realizar operaciones matemáticas
Terminar la instancia periódica
}

Función que maneja el evento de perdida de una meta
{ Ciclo de 1 hasta 100
Realizar operaciones matemáticas
Terminar la instancia periódica
}

Cuerpo Principal de TP
{
Informar que el evento de overload se manejará con una determinada función
Informar que el evento de perdida de meta se manejará con una determinada función

Informar que es una tarea periódica con el C_i y T_i pasados como parámetros de entrada y 0 para el tiempo de activación

Obtener la hora actual en C
Obtener la hora actual en D
Ciclar hasta que C + Ci >= D

Realizar cálculos matemáticos
Obtener la hora actual en D
Terminar la instancia periódica
}

9.2. TA- Tarea Aperiódica por Default

Cuerpo principal de TA

{

Informar que es una tarea aperiódica con el C_i y T_i pasados como parámetros de entrada

Obtener la hora actual en C

Obtener la hora actual en D

Ciclar hasta que C + Ci >= D

    Realizar cálculos matemáticos

    Obtener la hora actual en D

}

9.3. TK- Tarea que Define el Tiempo de Ejecución

Cuerpo principal de la rutina TK

{

set_aper(22000,2000,20000);

Informar que es una tarea aperiódica con el C_i= 2000 y T_i= 22000 y tiempo de activación = 22000

Llamar a la rutina que mata a las tareas periódicas

}

9.4. HS – Herramienta de Benchmark

```
Cuerpo principal de HS
{

Leer el archivo de entrada con los procesos

Ciclar por cada lote de prueba
{
        Generar un archivo de salida
        Correr Experimento 1
        Correr Experimento 2
        Correr Experimento 3
}
Experimento 1
{
 /* Prepara el experimento 1, decrementa en 100 el Ti de la tarea de Ti */
/* mas grande por cada ciclo                                            */

Busco la Tarea con Mayor Ti
Repetir N veces
  {
        Lanzar tareas
        Restarle 100 al Ti de la tarea más grande,
  }
}

Experimento 2
{
/* Prepara el experimento 2, decrementa en 1 el Ti de todas las tareas */
/* por cada ciclo                                                      */

Repetir N veces
   {
```

```
        restar 10 al Ti de cada tarea del lote de pruebas
        Lanzar Tareas
    }
}
Experimento 3
{
/* Prepara el experimento 3, agrega una tarea por cada ciclo con los valores */
/* de la ultima tarea                                                        */

Busca la ultima tarea
Lanzar Tareas
Repetir N veces
  {
   Generar una tarea nueva y agregarla al lote
    Lanzar Tareas
  }
}

Lanzar Tareas
{
Ciclar por todos los algoritmos
        {
                Limpiar las variables estadísticas
                Lanzar la tarea TK (que mata las tareas periódicas)
                Setear el algoritmo de planificación a usar

                Ciclar por cada tarea del lote de prueba
                  {
                        Lanzar una tarea periódica o aperiódica según corresponda
                  }
                Espero a que se terminen de ejecutar las tareas que lanze
                Escribo los resultados
        }
}
```

Apéndice B

Código

10. APENDICE B - CODIGO

El Código esta contenido en el diskette que se presenta con el trabajo. Cada directorio guarda relación con los directorios de Minix cuando esta instalado, y además se presenta el directorio HS que es donde esta implementado la herramienta de Benchmark.

Apéndice C

Casos de Prueba

11. APENDICE C: CASOS DE PRUEBA

Descripción de los casos de prueba

Caso	Verifica límite Teórico			Tareas		Observaciones
	Tasa Monotónica	Deadline Driven	Least Laxity First	Periódicas	Aperiódicas	
001	Si	Si	Si	Si	No	
002	No	No	No	Si	1 corta	El conjunto de tareas periódicas verifica el límite de Tasa Monotónica. Permite ver la interferencia de una tarea aperiódica en un conjunto planificable.
003	No	No	No	Si	5 cortas	El conjunto de tareas periódicas verifica el límite de Tasa Monotónica. Se crea una interferencia mayor que en el lote 2 para el conjunto planificable.
004	No	No	No	Si	1 muy larga	El conjunto de tareas periódicas verifica el límite de Tasa Monotónica. Se logra una interferencia constante al conjunto planificable.
005	No	No	No	Si	1 muy larga y 3 cortas	El conjunto de tareas periódicas verifica el límite de Tasa Monotónica. A la interferencia constante se la agrega una carga adicional
006	No	No	No	No	Si	Una de las tareas es muy larga. Se testea la habilidad de los algoritmos para planificar tareas aperiódicas
007	Si	Si	Si	No	1 larga y	Se testea la habilidad de los algoritmos para planificar tareas aperiódicas.

					una corta	
008	No	No	No	No	Si	Una tarea muy larga y una con tiempo de ejecución muy cerca de la meta. Haciendo al lote de prueba no planificable.
009	No	No	No	1	Si y cortas	Este lote presenta mucho trabajo de para el procesador al principio, hasta que se cumplan todas las tareas aperiódicas y luego el trabajo se disminuye sensiblemente.
010	No	No	No	Si	Si	La única tarea periódica del lote tiene tiempo de ejecución muy cerca de la meta. Este lote es similar al anterior, pero la tarea periódica casi no admite interferencias para cumplir su meta
011	Si	Si	Si	Si, armónicas	No	Este lote testea la habilidad de Tasa Monotónica para planificar tareas periódicas.
012	No	No	No	Si, armónicas	Si, armónicas	El conjunto de tareas periódicas verifica el límite teórico de Tasa Monotónica la tarea aperiódica es armónica con las demás tareas. Este lote prueba la habilidad del algoritmo para planificar tareas armónicas periódicas con una interferencia también armónica
013	No	No	No	Si, armónicas		
014	Si	Si	Si	Si	No	Las tareas son cortas e iguales. Testean la habilidad de los algoritmos de planificar lotes grandes de prueba.
015	Si	Si	Si	No	Si	Las tareas son cortas e iguales. Testean la habilidad de los algoritmos de planificar lotes grandes de prueba.
016	Si	Si	Si	Si	No	Los C_i y T_i de las tareas son muy chicos. Testea los algoritmos con muchos cambios de contexto.

017	Si	Si	Si	Si	Si	Los C_i son casi inexistentes. Testea los algoritmos con muchos cambios de contexto
018	No	No	No	Si	Si	La tarea aperiódica tiene el C_i es casi igual a T_i. En este caso los algoritmos debe planificar al principio un conjunto de tareas pesadas y luego termina con una sola periódica que no admite intervenciones del sistema operativo.
019	No	No	No	Si	Si	Las tareas son muy largas, algunas de ellas con el peor tiempo de ejecución igual a la meta
020	No	No	No	No	Si	EL tiempo de ejecución de una tarea sobrepasa la meta. Esto testea el comportamiento de los algoritmos ante errores del sistema
021	Si	Si	Si	Si	Si	
022	No	No	No	Si	Si	Este conjunto sobrepasa por mucho el límite teórico de los algoritmos. Testea los algoritmos ante situaciones extremas
023	No	No	No	Si, muy cortas	No	El tiempo de ejecución de las tareas es casi igual a la meta.
024	No	No	No	No	Si, muy cortas	El tiempo de ejecución de las tareas es casi igual a la meta.
025	No	No	No	No	Si	El tiempo de ejecución excede la meta en todas las tareas.
026	No	No	No	Si	Si	Cada tarea no permite interferencia de ninguna clase para llegar a su meta.

Se define una tarea "larga" cuando la meta de la misma es mucho mas grande que el quantum del procesador. Se define una tarea "corta" cuando la meta de la misma es mucho menor que el quantum del procesador.

Casos de prueba

Caso 1 : P(100,20).P(150,40).P(350,100);

Caso 2 : P(100,20).P(150,40).P(350,100).A(300,100);

Caso 3 : P(100,20).P(150,40).P(350,100).A(300,100).A(300,10).A(300,50).A(200,10).A(100,10);

Caso 4 : P(100,20).P(150,40).P(350,100).A(1300,800);

Caso 5 : P(100,20).P(150,40).P(350,100).A(1300,800).A(100,10).A(200,20).A(200,40);

Caso 6 : A(1300,800).A(100,10).A(200,30).A(300,20).A(210,50).A(300,10);

Caso 7 : A(1300,800).A(100,10);

Caso 8 : A(1300,800).A(100,90);

Caso 9 : A(300,80).A(100,90).A(100,20).A(400,30).A(100,10).A(200,50).P(100,40);

Caso 10 : A(300,80).A(100,90).A(100,20).A(400,30).A(100,10).A(200,50).P(100,90);

Caso 11 : P(200,20).P(200,40).P(200,80);

Caso 12 : P(200,20).P(200,40).P(200,80).A(200,160);

Caso 13 : P(200,40).P(200,80).P(200,160);

Caso 14 : P(200,20).P(200,20).P(200,20).P(200,20).P(200,20).P(200,20).P(200,20);

Caso 15 : A(200,20).A(200,20).A(200,20).A(200,20).A(200,20).A(200,20).A(200,20).A(200,20).A(200,20);

Caso 16 : P(20,5).P(10,3).P(40,9);

Caso 17 : P(40,1).P(50,1).P(20,1).A(10,1).P(100,1).A(40,2);

Caso 18 : A(400,150).A(500,130).P(200,190).A(200,180).A(200,170);

Caso 19 : A(4000,1500).P(5000,3000).A(2000,1900).A(2000,2000).A(1700,1700);

Caso 20 : A(1000,100).A(400,300).A(300,700).A(500,50).A(600,110);

Caso 21 : P(10,1).P(40,3).P(30,7).P(50,5).A(60,11);

Caso 22 : P(40,150).A(5,1).P(20,190).A(2,2).A(1,4);

Caso 23 : P(40,15).P(50,13).P(20,19);

Caso 24 : A(40,15).A(50,13).A(20,19);

Caso 25 : A(2,10).A(4,40).A(8,30).A(6,50).A(12,60);

Caso 26 : A(11,10).P(41,40).A(31,30).P(51,50).A(61,60);

Apéndice D

Gráficos

Lotes de Prueba

A continuación se presentan los gráficos con la relación entre el C_i y T_i de cada tarea del lote de prueba así como también una comparación del límite del conjunto de prueba con el límite teórico de Tasa monotónica, Deadline Driven y Least Laxity First. El límite teórico de estos dos últimos algoritmos se presenta junto porque es el mismo.

Note que la relación entre el C_i y T_i de cada tarea se muestra como un gráfico de columnas apiladas donde el C_i esta abajo del T_i. En este tipo de gráfico se nota claramente la proporción de uno con respecto al otro. También se diferenció claramente las tareas aperiódicas del conjunto de las periódicas por el código de colores indicado más abajo.

La primera columna de gráficos representa los casos de prueba y la segunda la relación de los límites de cada lote de prueba.

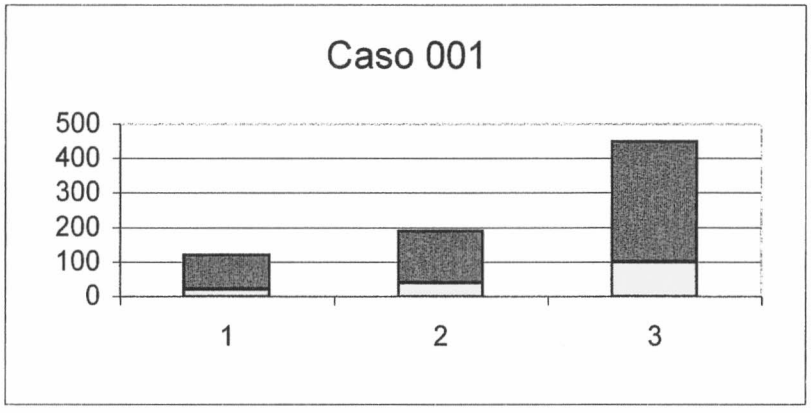
Referencias:

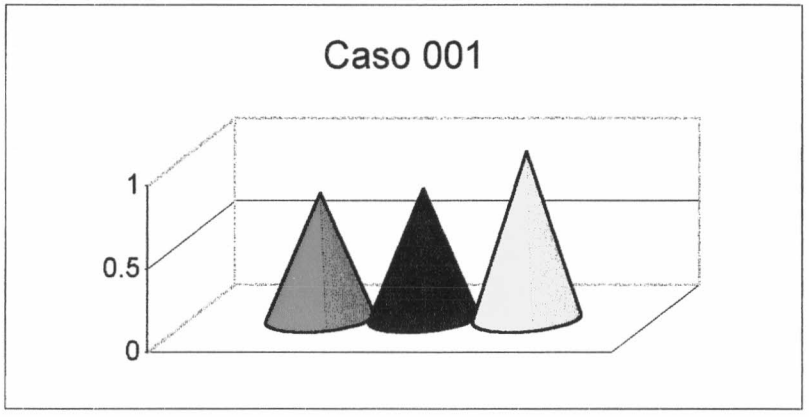
Casos de Prueba

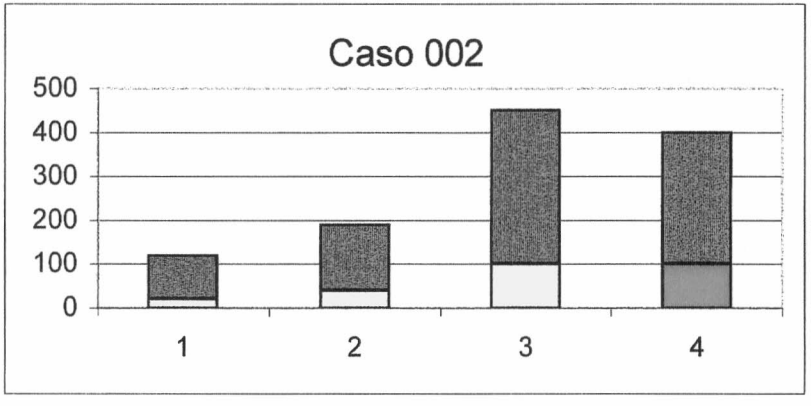
- C_i de una tarea periódica
- T_i de una tarea periódica
- C_i de una tarea aperiódica
- T_i de una tarea aperiódica

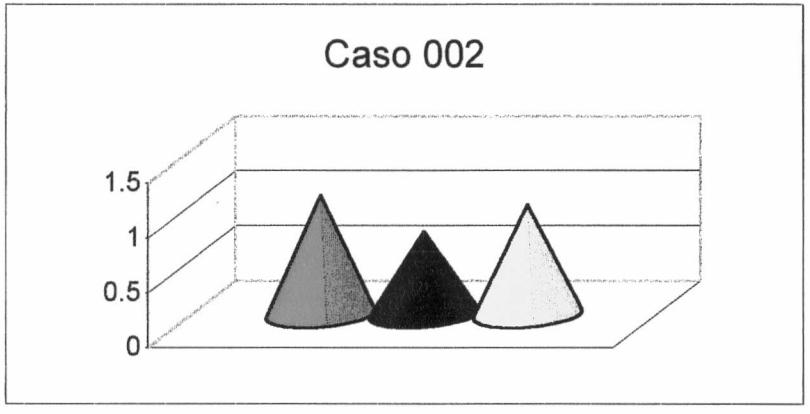
Límite

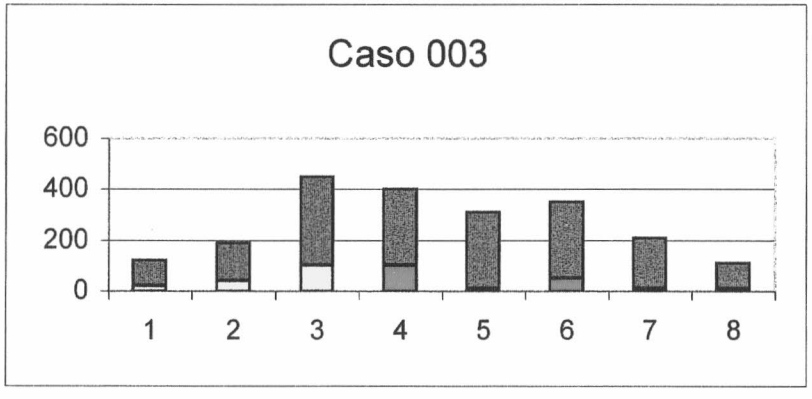
- Límite del lote de prueba
- Límite teórico de Tasa Monotónica
- Límite teórico de Deadline Driven y Least Laxity First

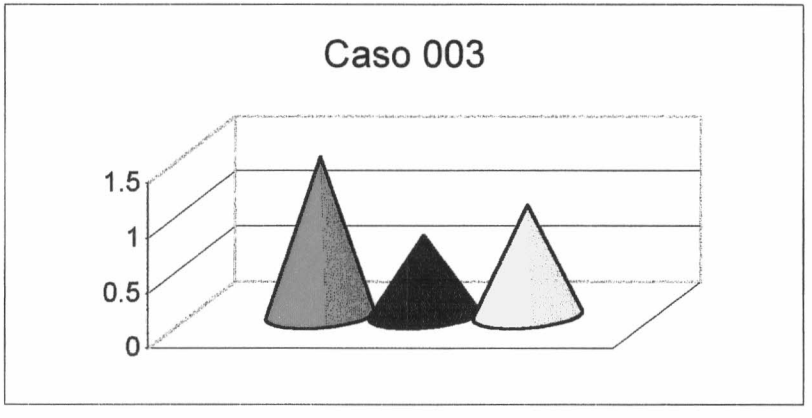

Caso 001
500
400
300
200
100
0
1
2
3

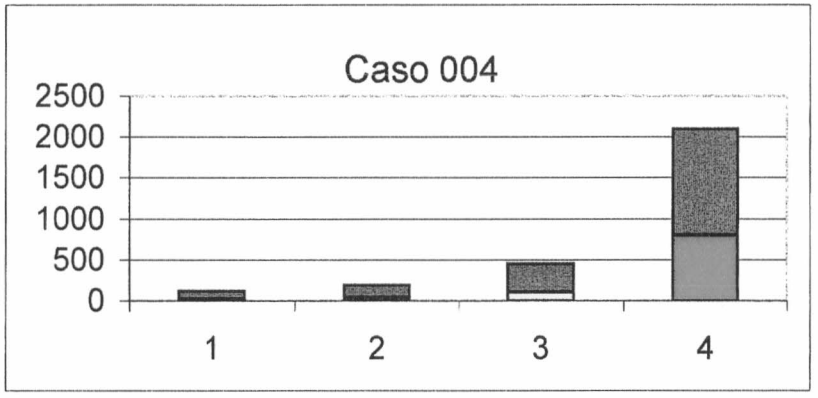

Caso 001
1
0.5
0

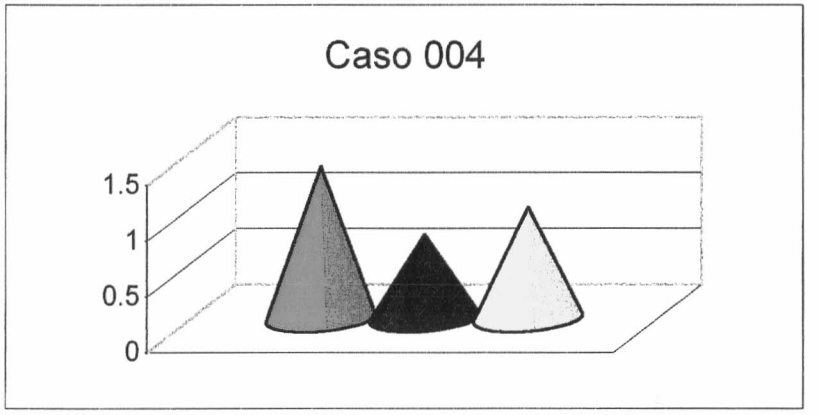

Caso 002
500
400
300
200
100
0
1
2
3
4

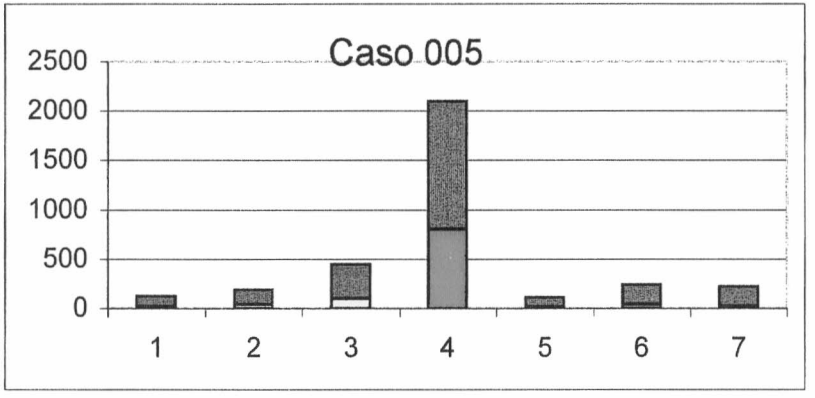

Caso 002
1.5
1
0.5
0

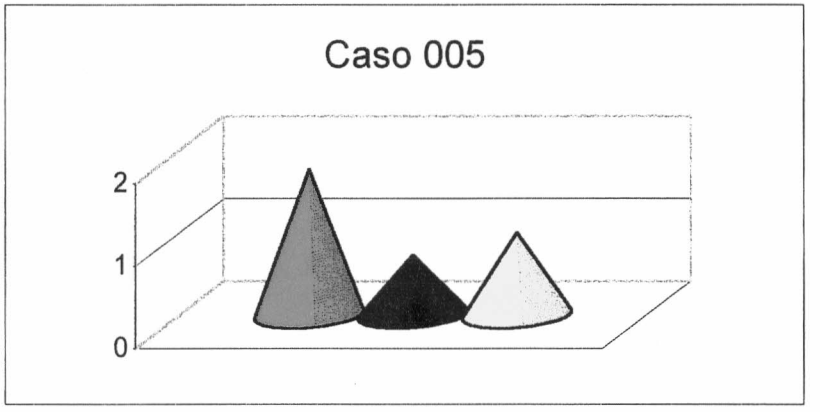

Caso 003
600
400
200
0
1
2
3
4
5
6
7
8

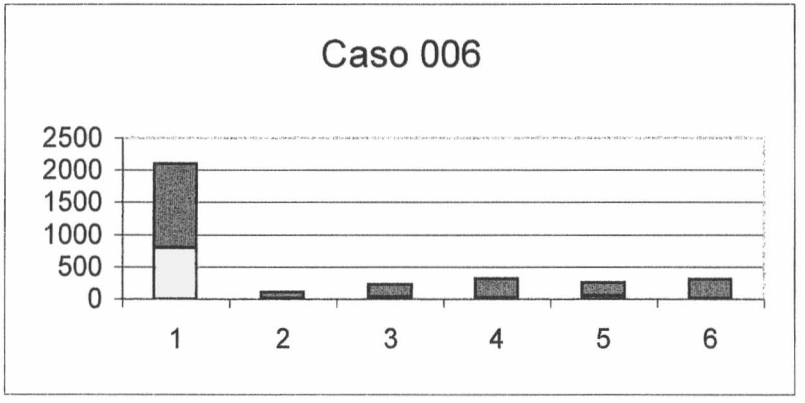

Caso 003
1.5
1
0.5
0

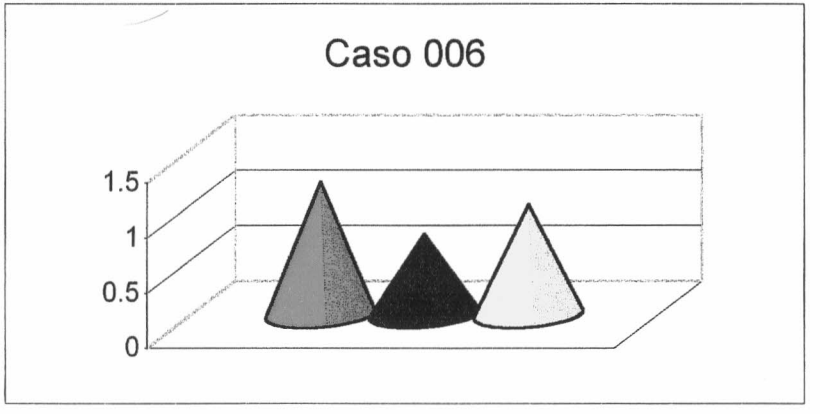

Caso 004
2500
2000
1500
1000
500
0
1
2
3
4

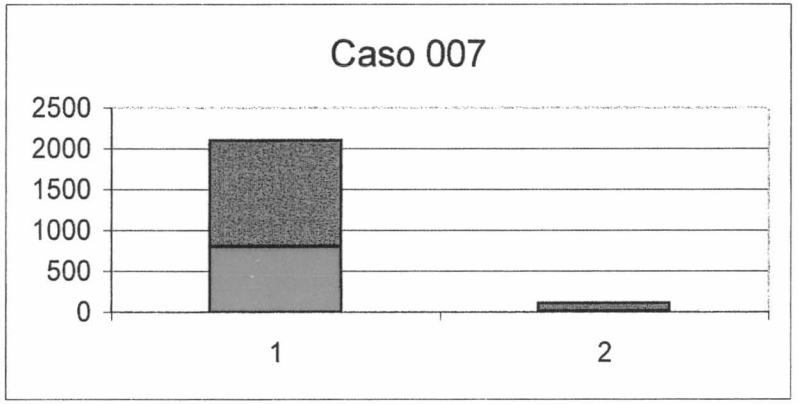

Caso 004
1.5
1
0.5
0

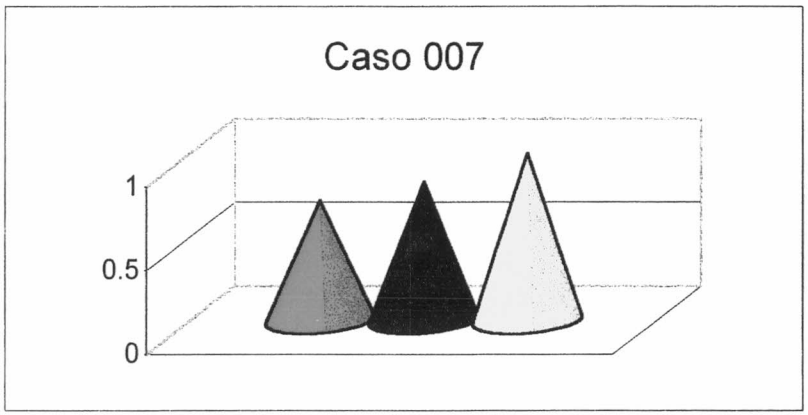

Caso 005
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7

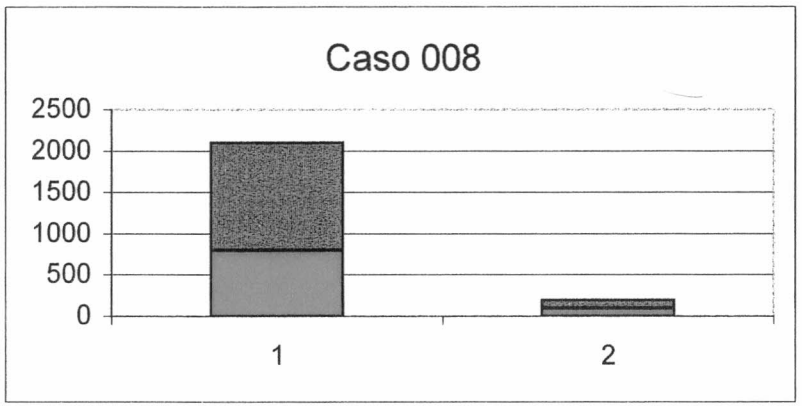

Caso 005
2
1
0

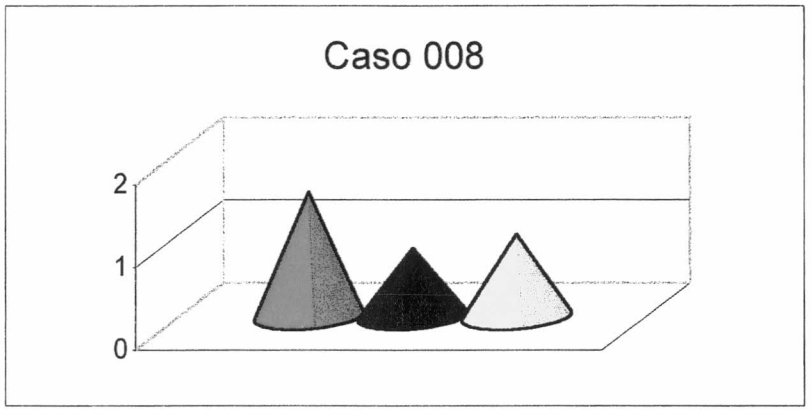

Caso 006
2500
2000
1500
1000
500
0
1
2
3
4
5
6

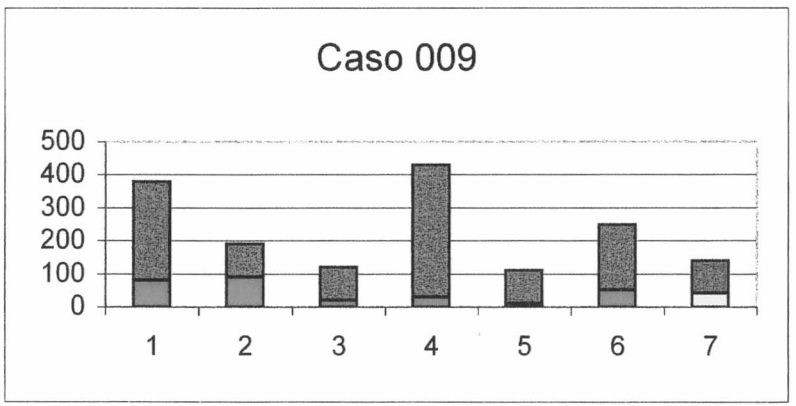

Caso 006
1.5
1
0.5
0

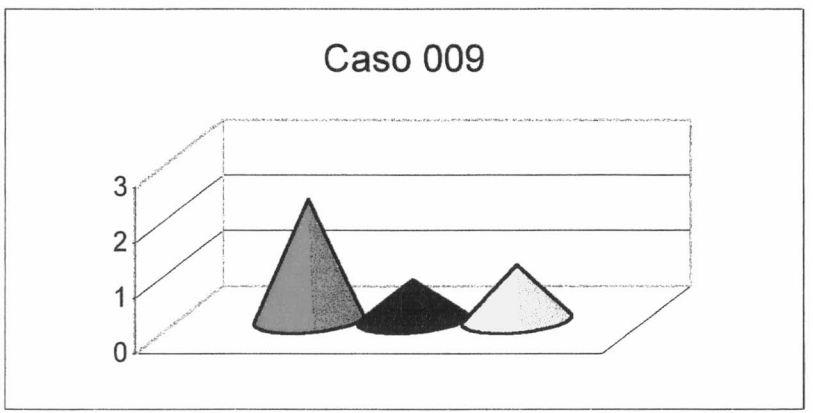

Caso 007
2500
2000
1500
1000
500
0
1
2

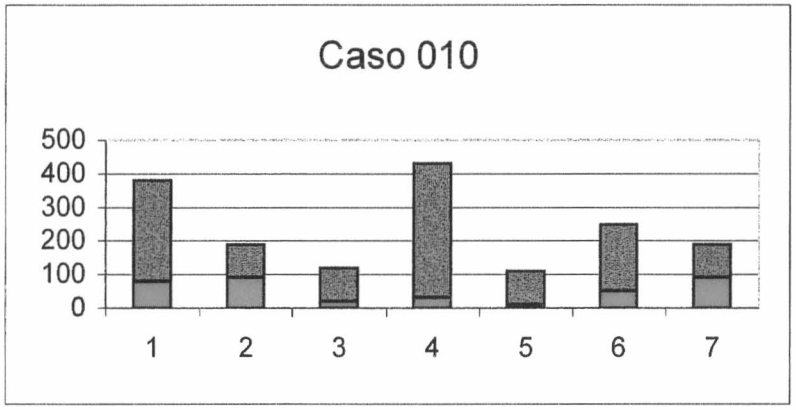

Caso 007
1
0.5
0

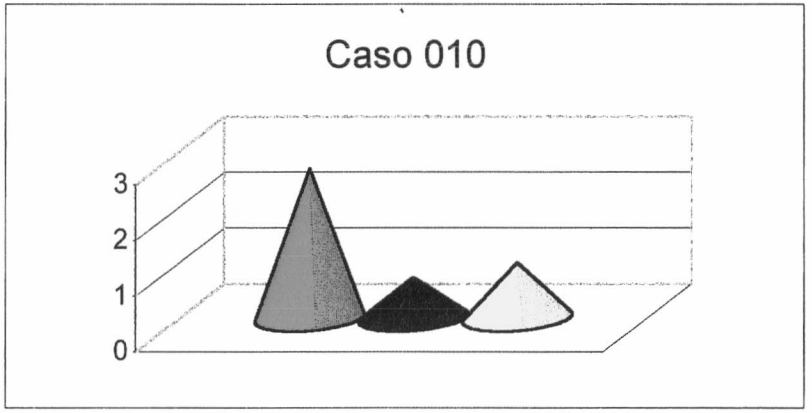

Caso 008
2500
2000
1500
1000
500
0
1
2

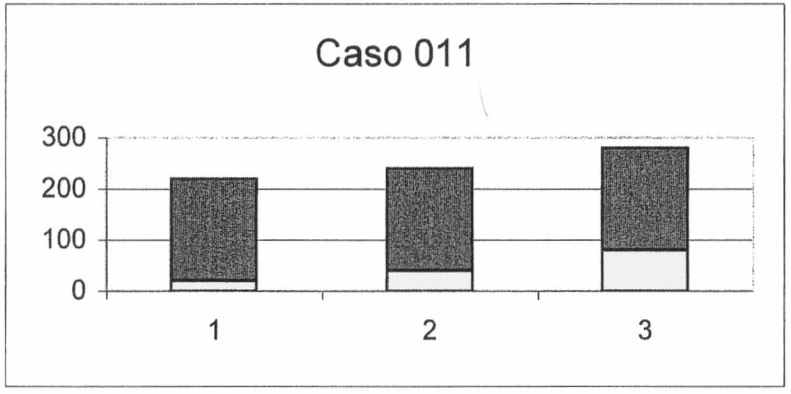

Caso 008
2
1
0

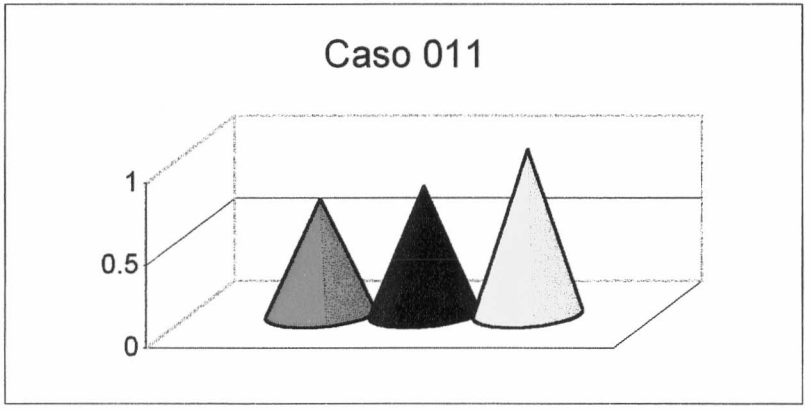

Caso 009
500
400
300
200
100
0
1
2
3
4
5
6
7

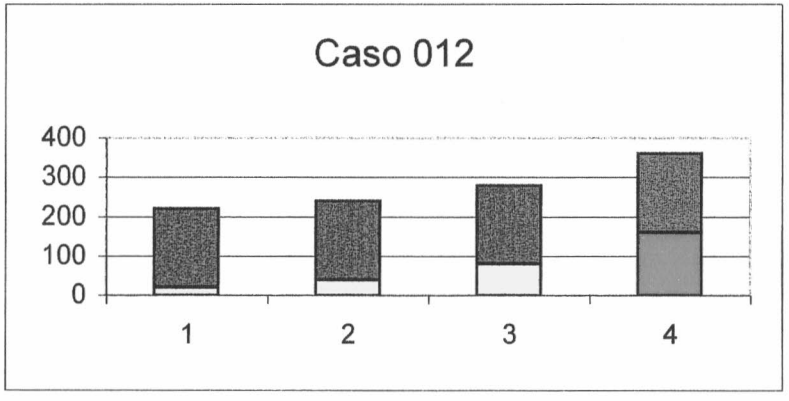

Caso 009
3
2
1
0

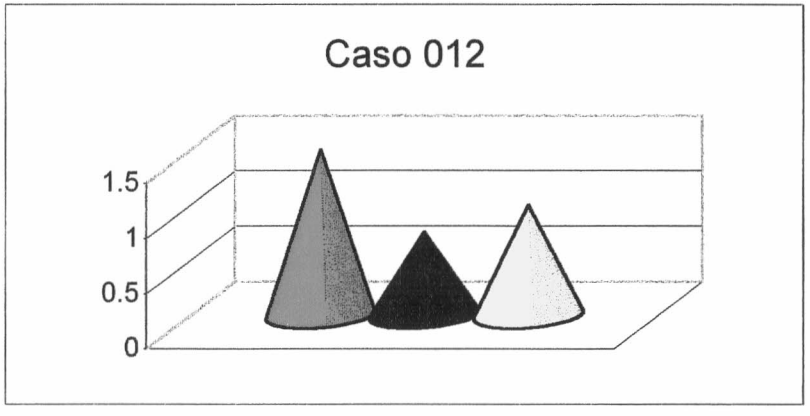

Caso 010
500
400
300
200
100
0
1
2
3
4
5
6
7

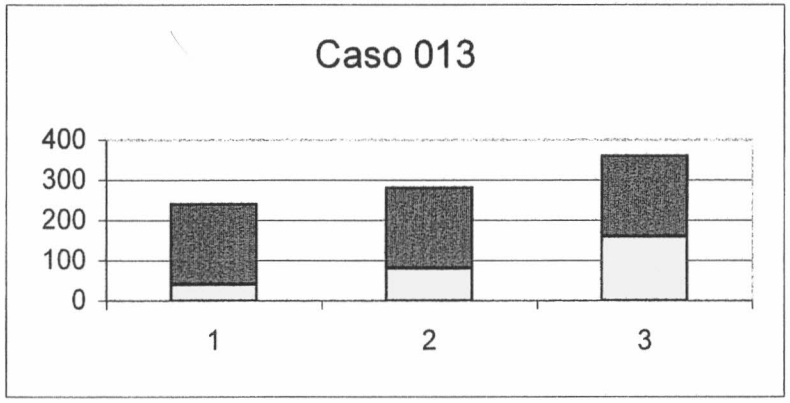

Caso 010
3
2
1
0

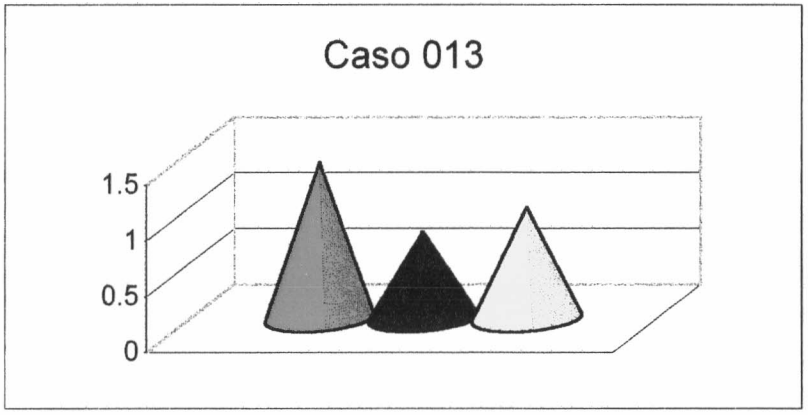

Caso 011
300
200
100
0
1
2
3

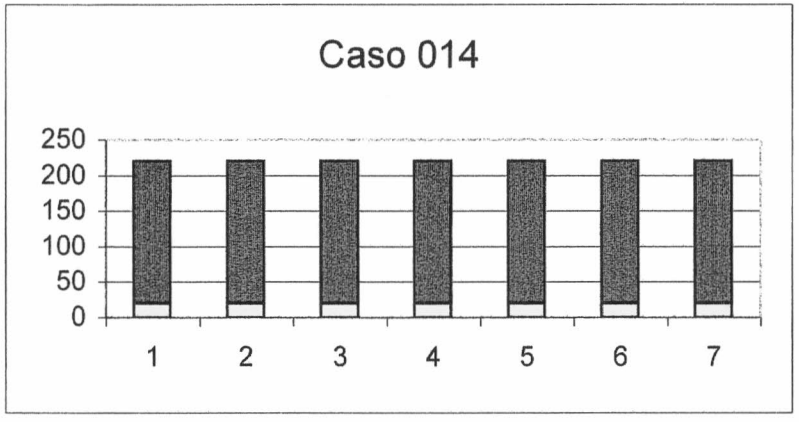

Caso 011
1
0.5
0
1
2
3

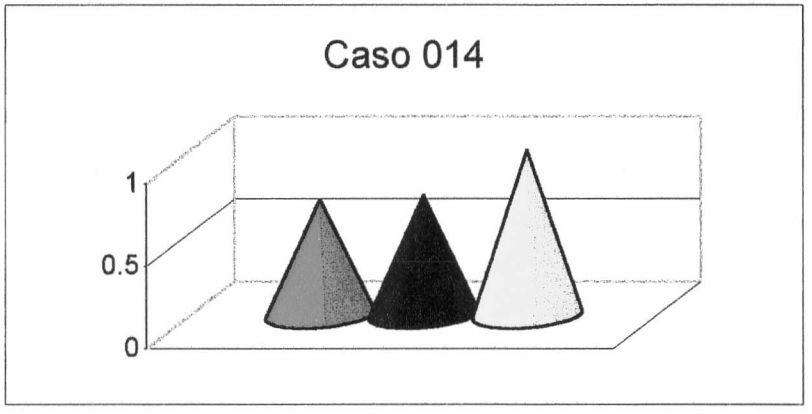

Caso 012
400
300
200
100
0
1
2
3
4

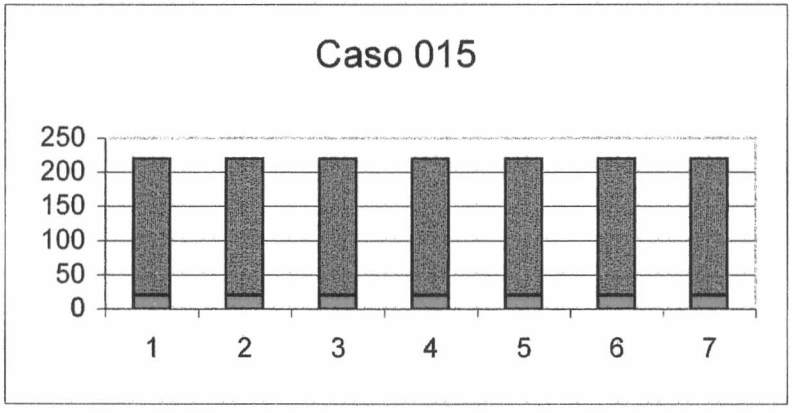

Caso 012
1.5
1
0.5
0

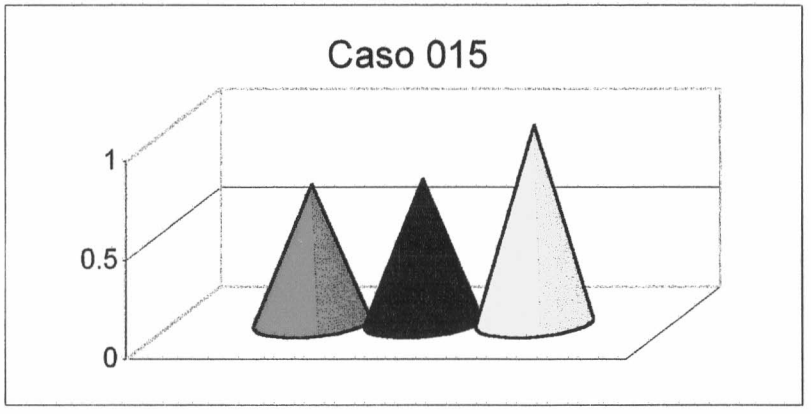

Caso 013
400
300
200
100
0
1
2
3

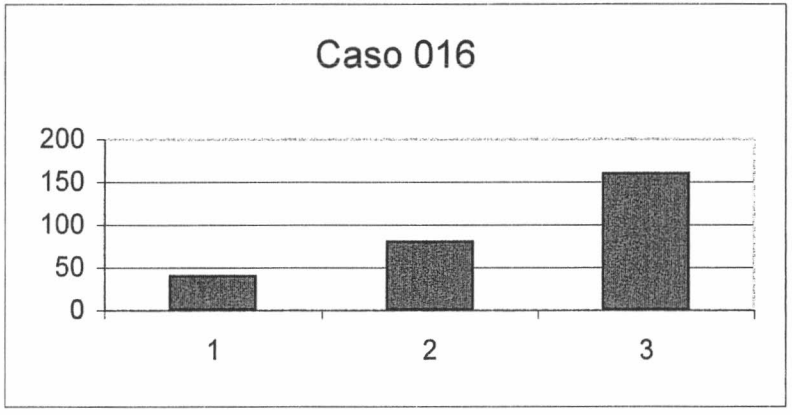

Caso 013
1.5
1
0.5
0

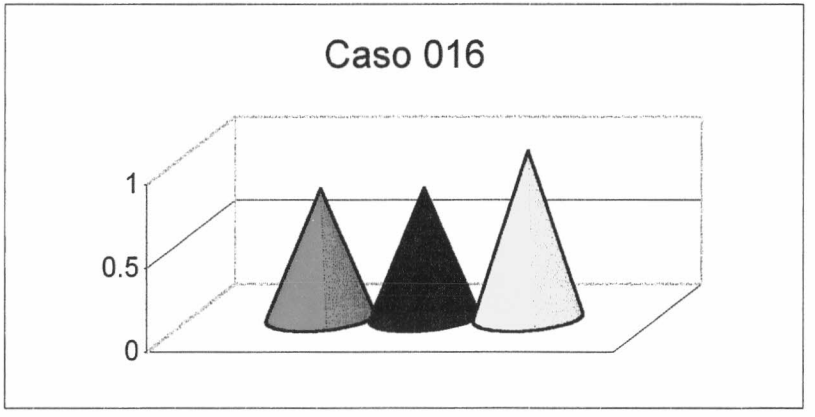

Caso 014
250
200
150
100
50
0
1
2
3
4
5
6
7

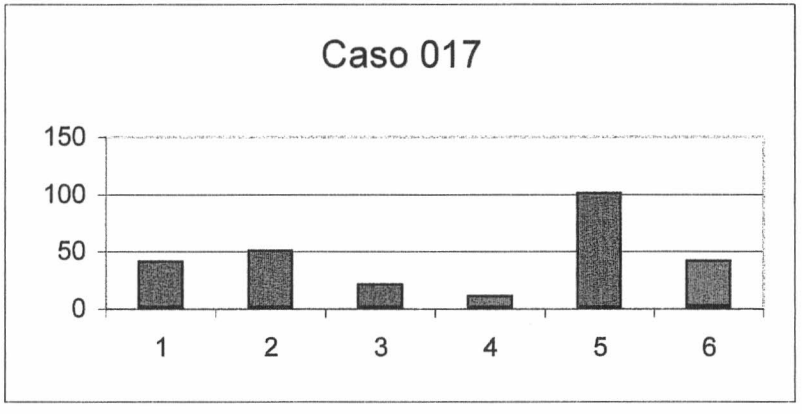

Caso 014
1
0.5
0

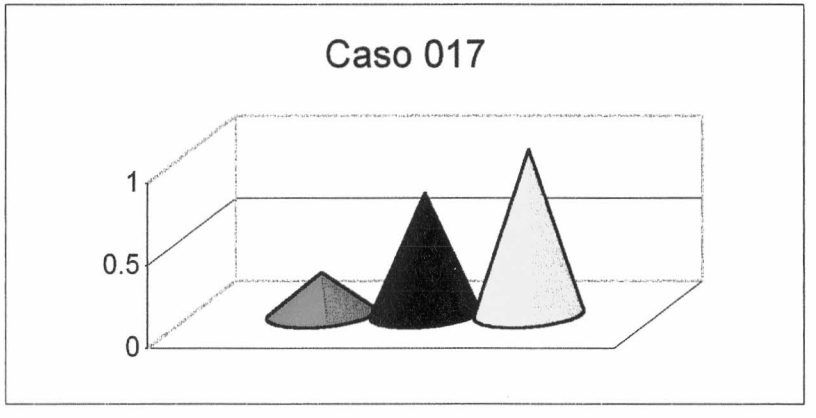

Caso 015
250
200
150
100
50
0
1
2
3
4
5
6
7

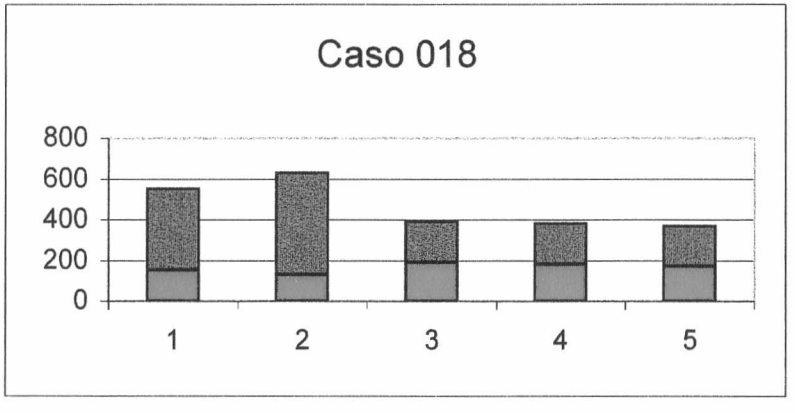

Caso 015
1
0.5
0

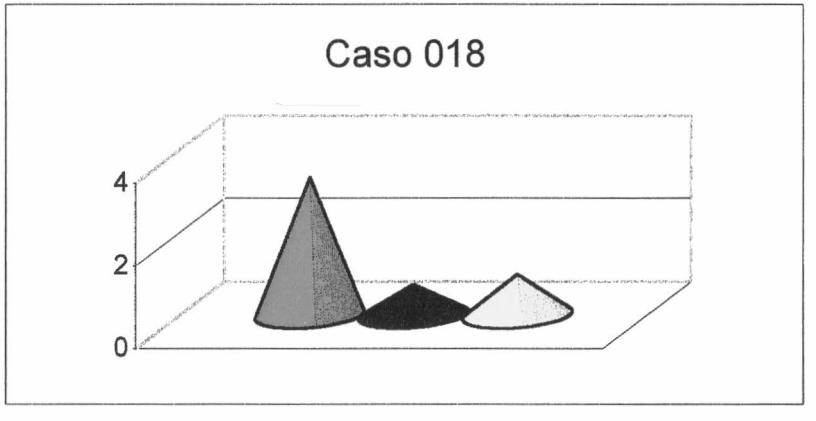

Caso 016
200
150
100
50
0
1
2
3

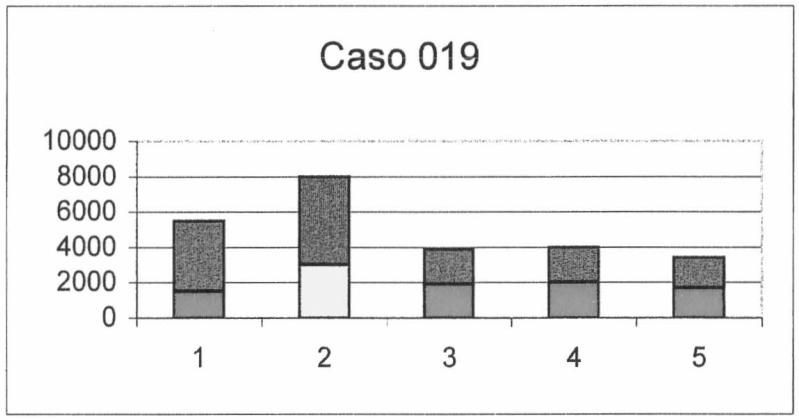

Caso 016
1
0.5
0

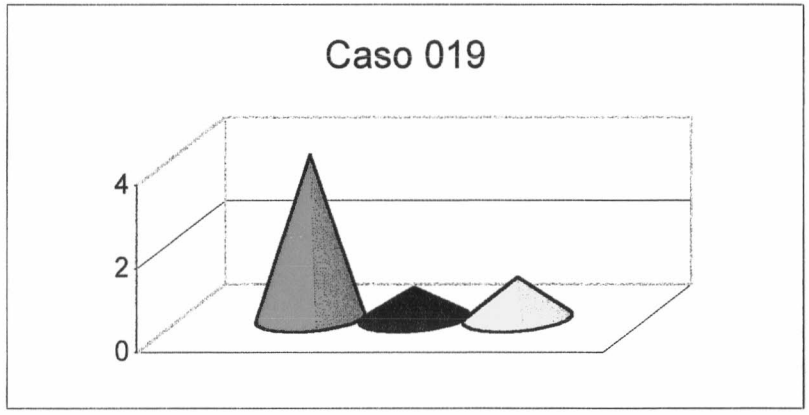

Caso 017
150
100
50
0
1
2
3
4
5
6

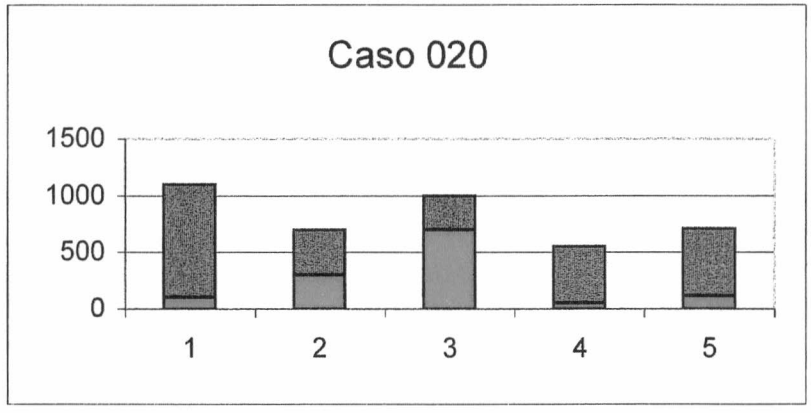

Caso 017
1
0.5
0

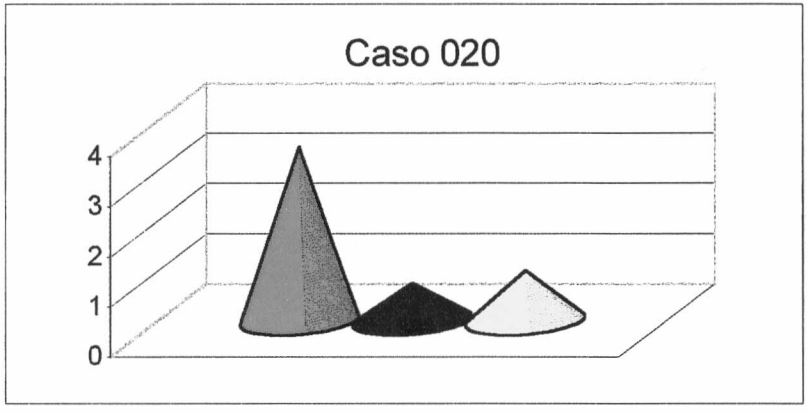

Caso 018
800
600
400
200
0
1
2
3
4
5

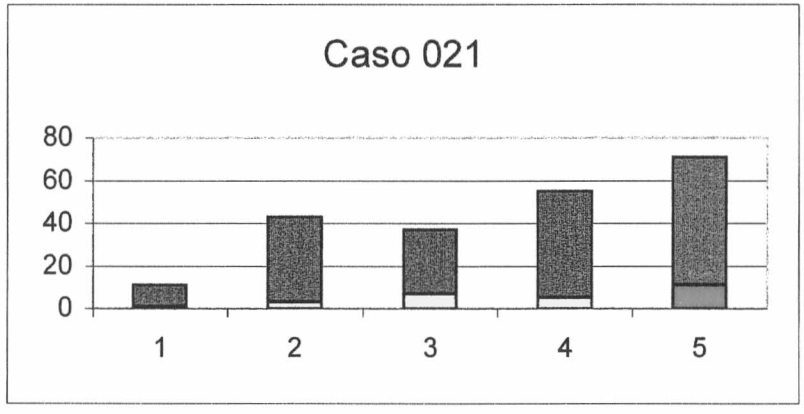

Caso 018
4
2
0

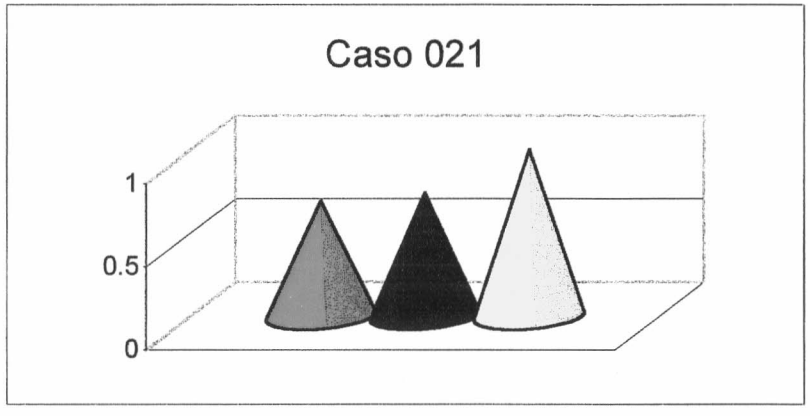

Caso 019
10000
8000
6000
4000
2000
0
1
2
3
4
5

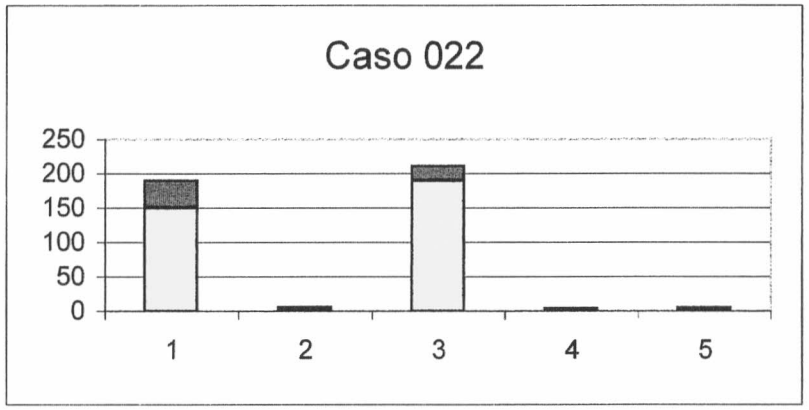

Caso 019
4
2
0

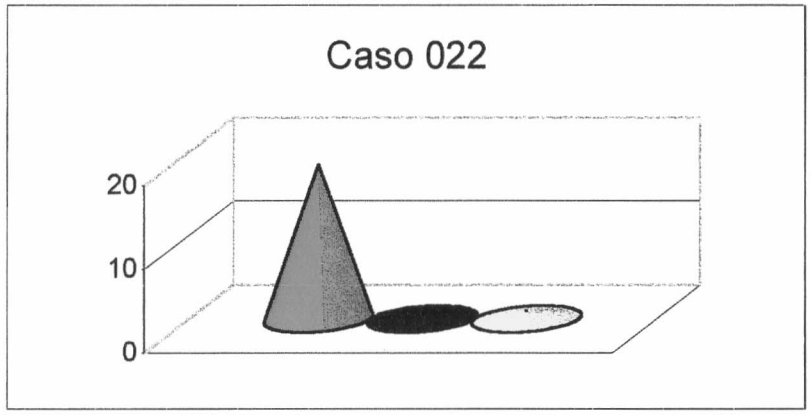

Caso 020
1500
1000
500
0
1
2
3
4
5

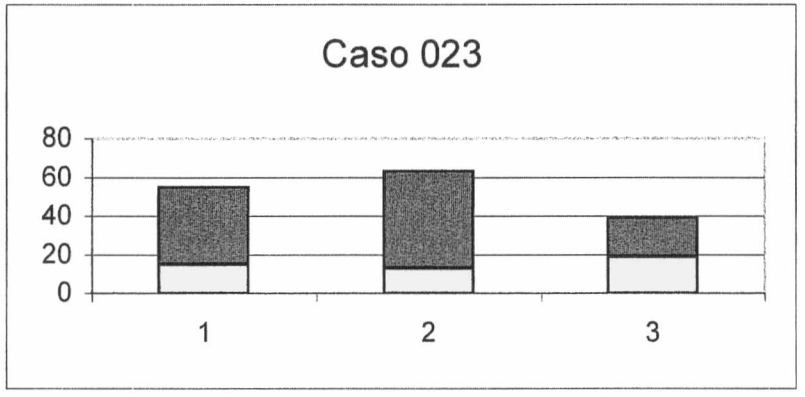

Caso 020
4
3
2
1
0

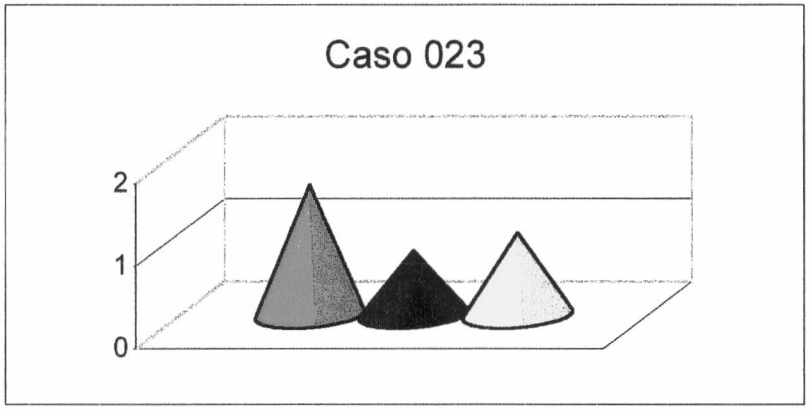

Caso 021
80
60
40
20
0
1
2
3
4
5

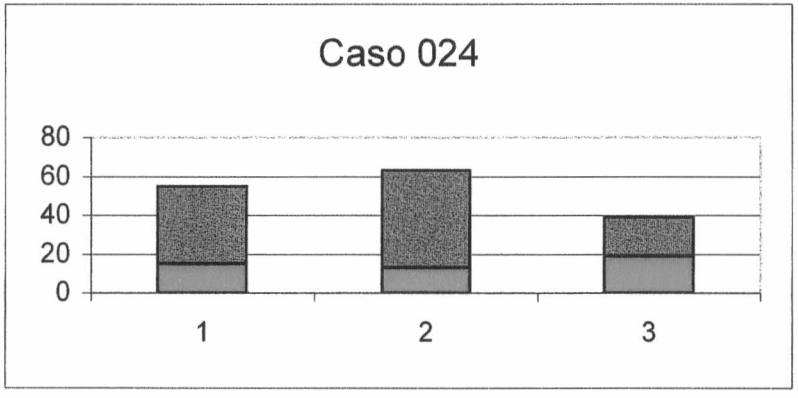

Caso 021
1
0.5
0

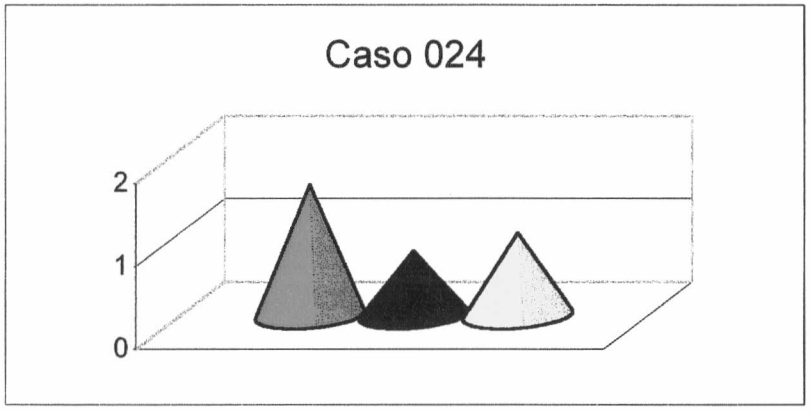

Caso 022
250
200
150
100
50
0
1
2
3
4
5

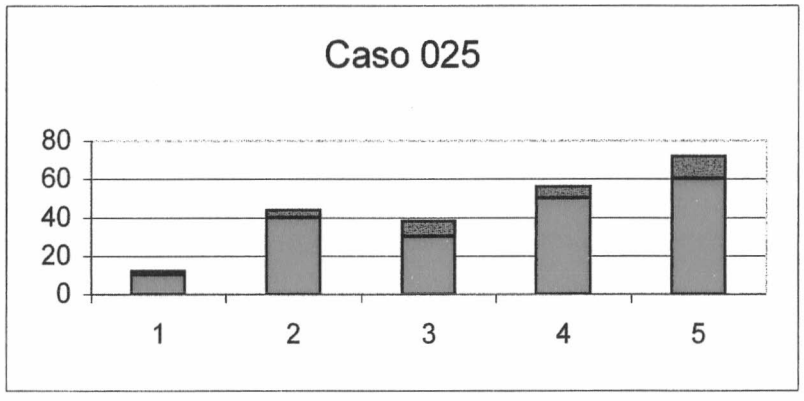

Caso 022
20
10
0

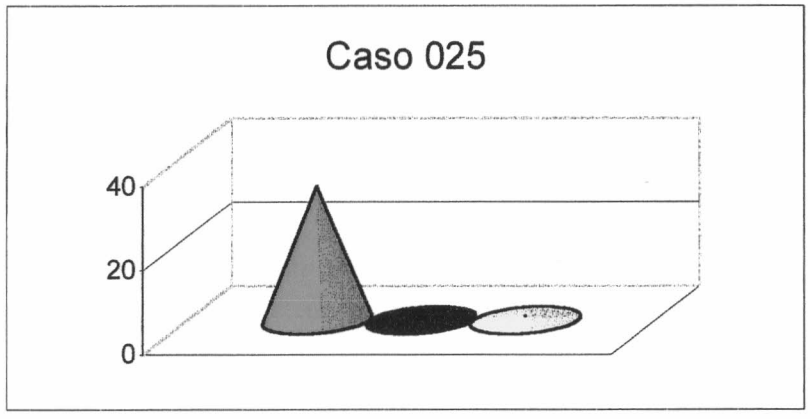

Caso 023
80
60
40
20
0
1
2
3

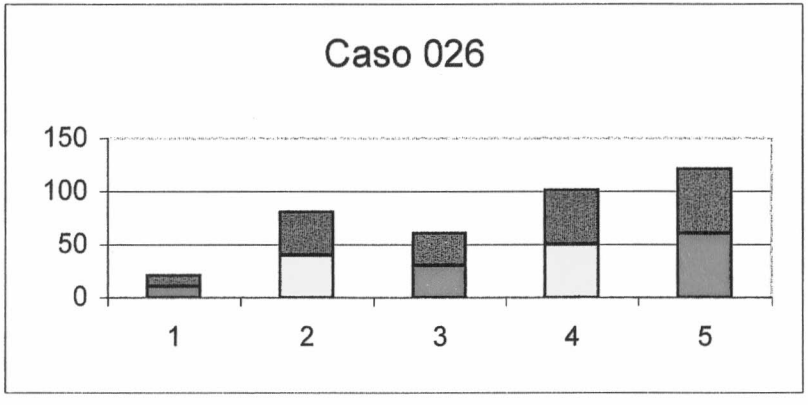

Caso 023
2
1
0

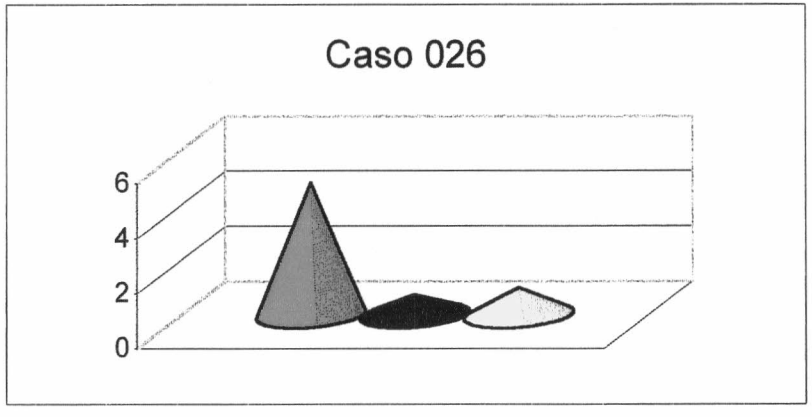

Caso 024
80
60
40
20
0
1
2
3


Caso 024
2
1
0


Caso 025
80
60
40
20
0
1
2
3
4
5


Caso 025
40
20
0


Caso 026
150
100
50
0
1
2
3
4
5

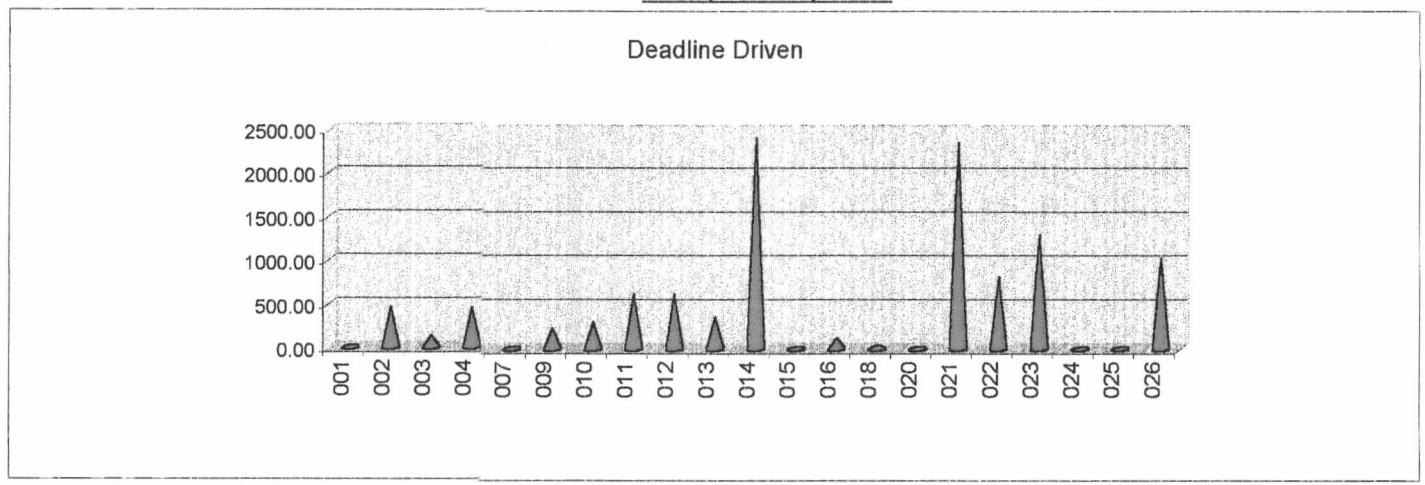

Caso 026
6
4
2
0

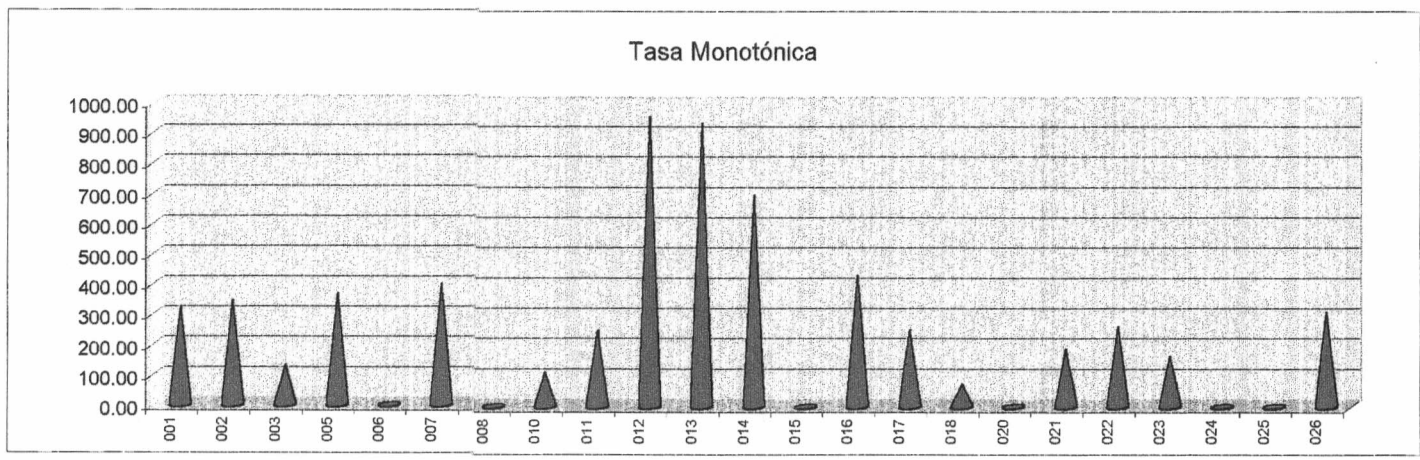
Algoritmos – Cantidad de metas perdidas

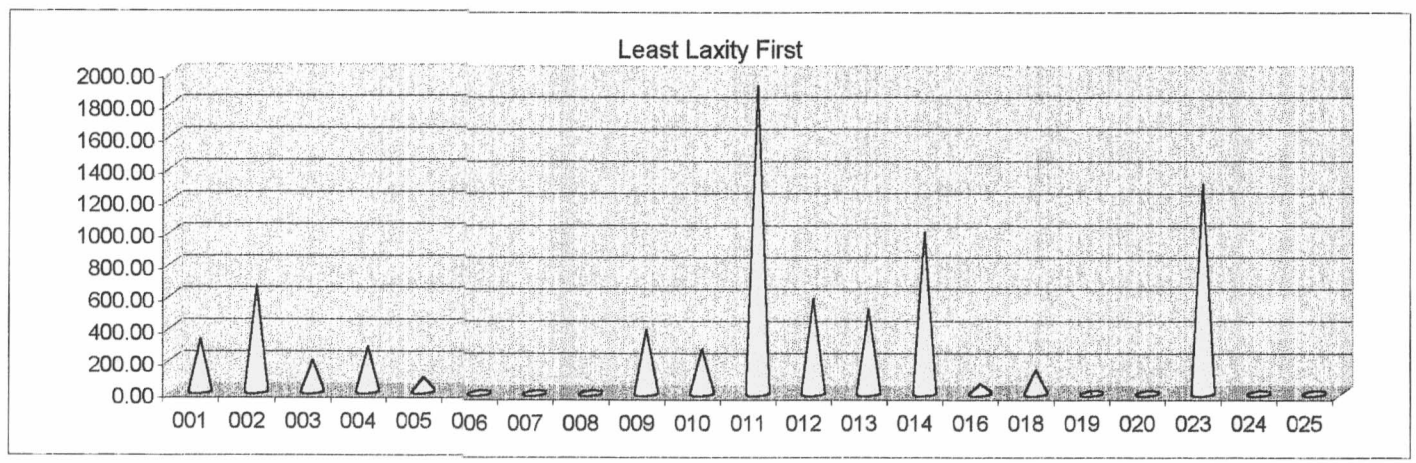
En los siguientes gráficos se muestra una comparación de la cantidad de metas perdidas, tanto periódicas como aperiódicas de los distintos algoritmos testeados. Se confeccionó un gráfico para las metas perdidas periódicas y otro para las aperiódicas.

Note que en los diferentes gráficos algunos lotes de prueba no aparecen. Esto se debe a que no se pudo obtener resultados para ese lote de prueba con ese algoritmo. No es lo mismo que cuando el valor de algún dato esta en cero (0) que indica que no se perdió ninguna meta. Esta aclaración es valida para todos los gráficos presentados.

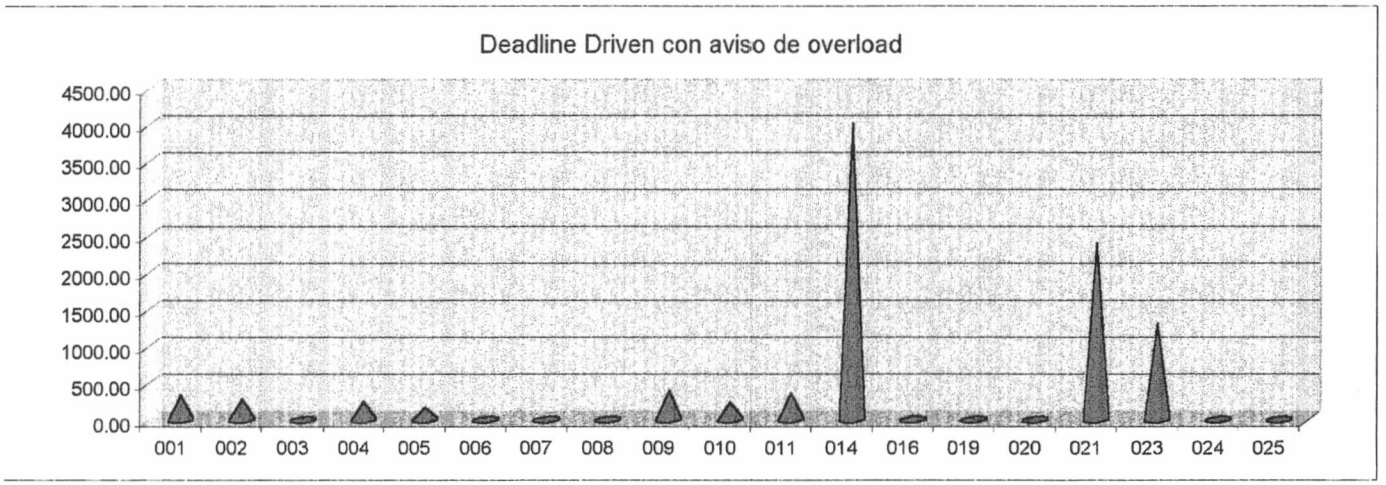
Metas periódicas perdidas

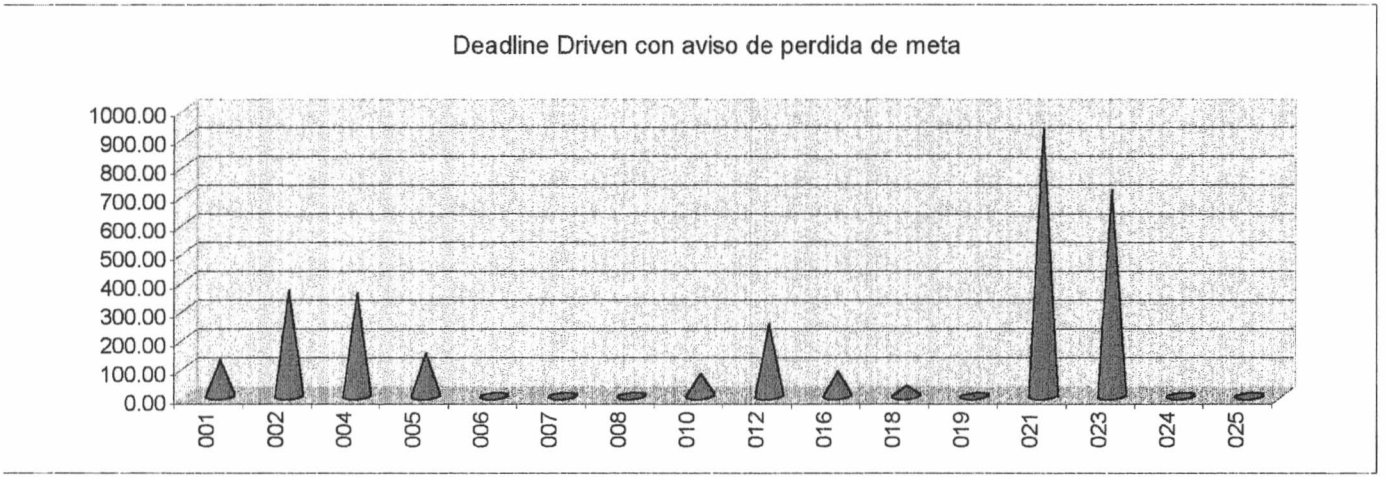


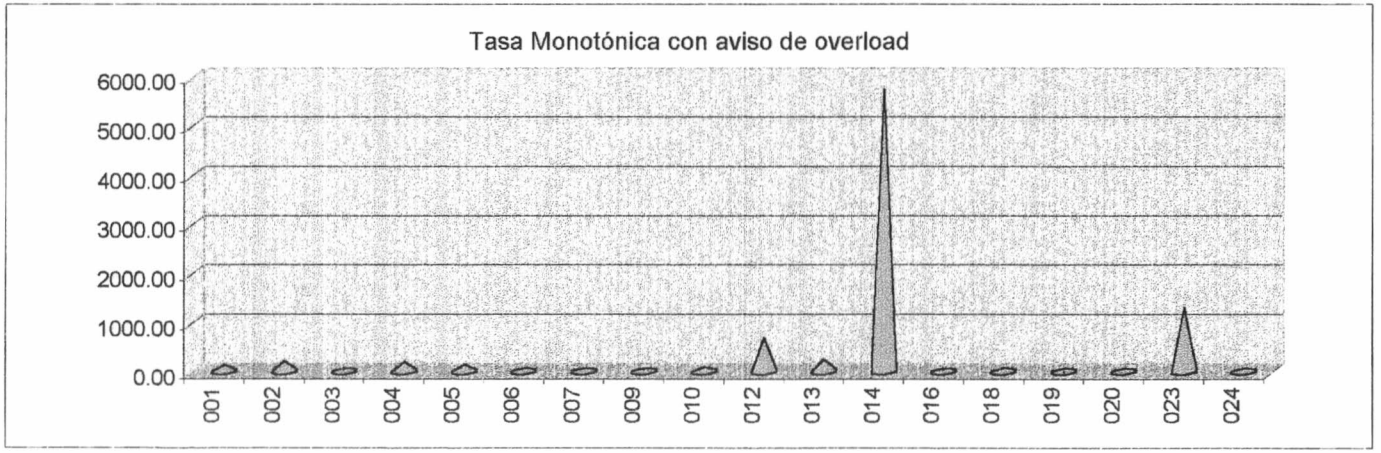




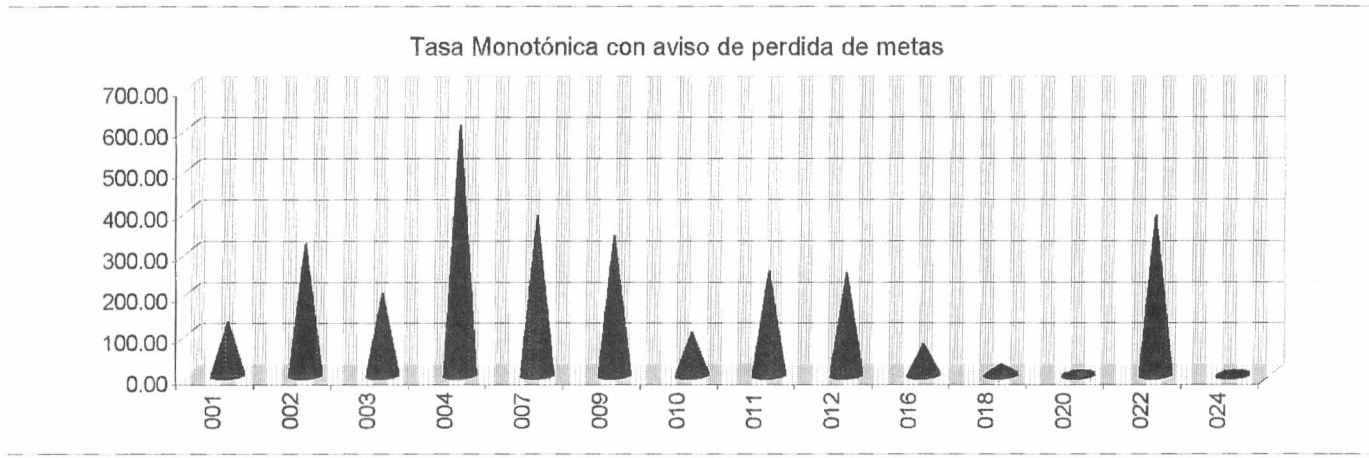
Metas periódicas perdidas



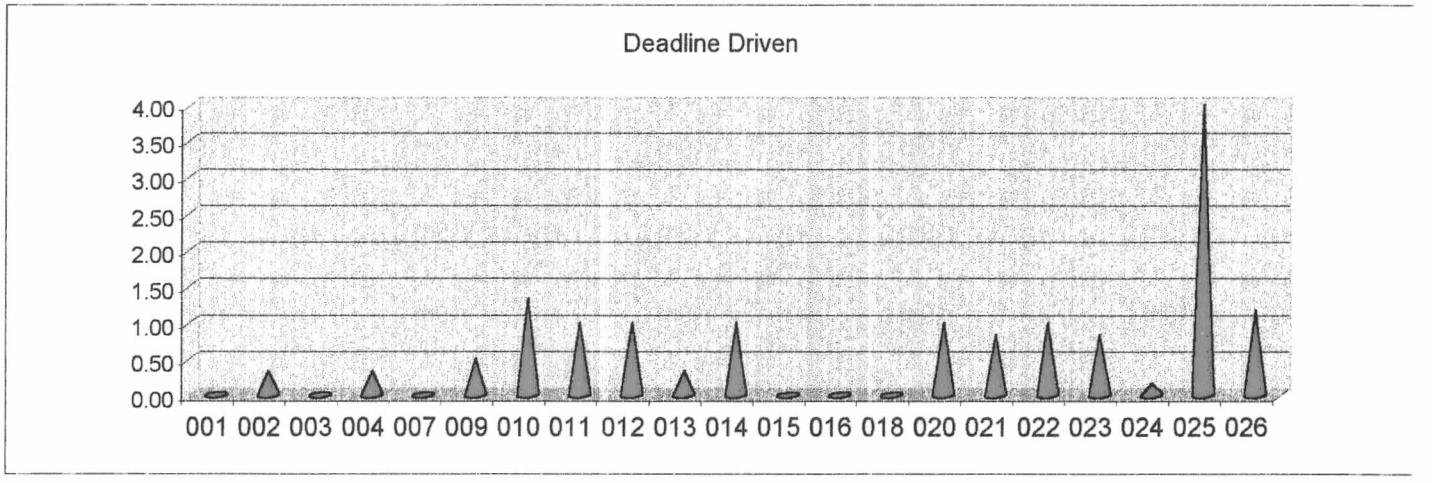


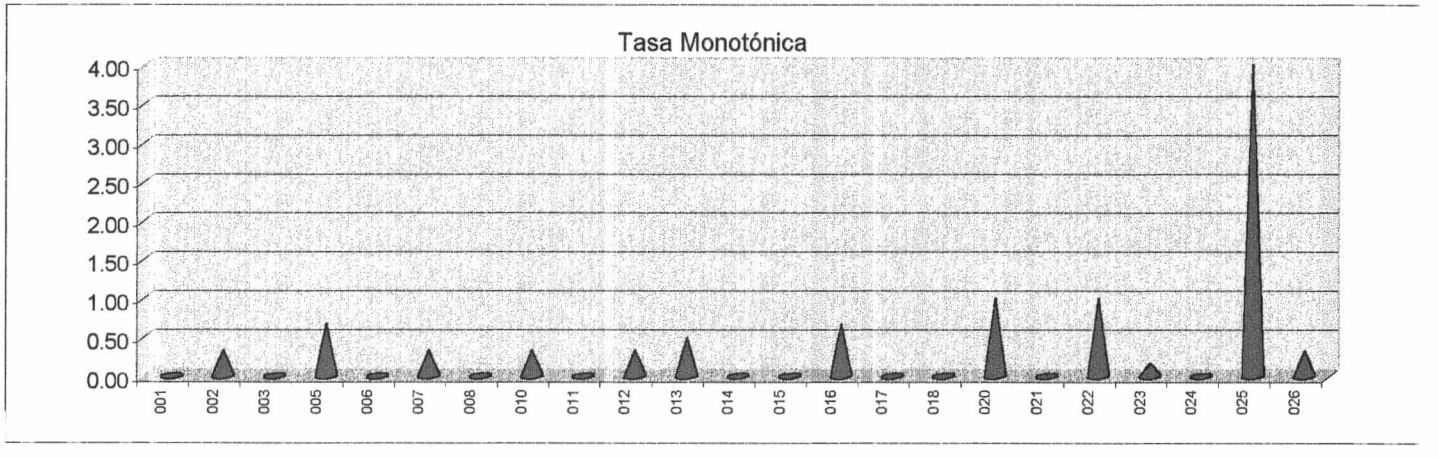


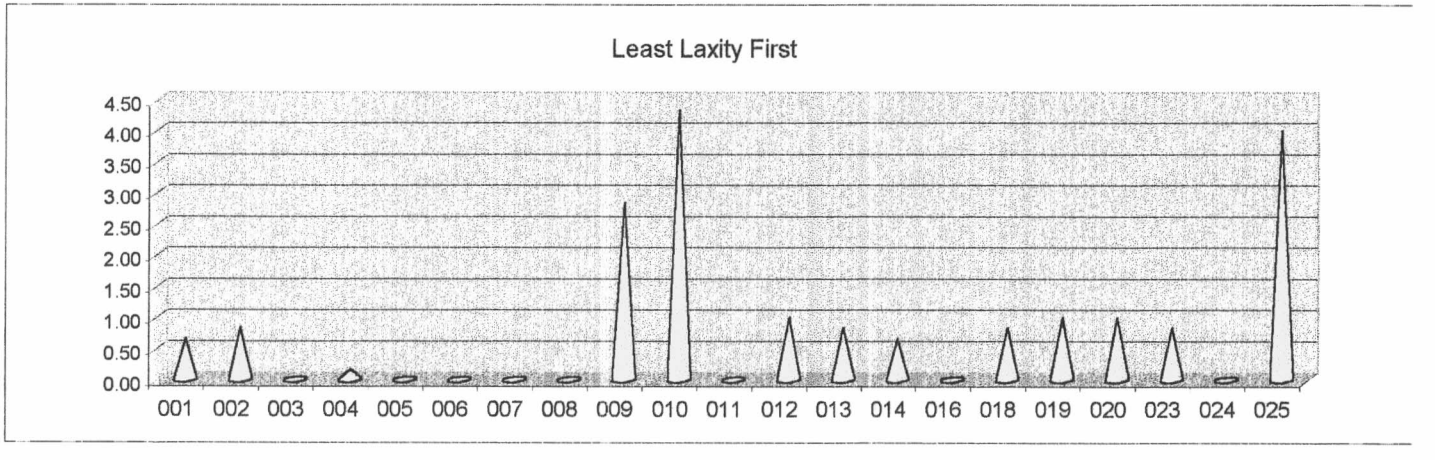
Metas periódicas perdidas



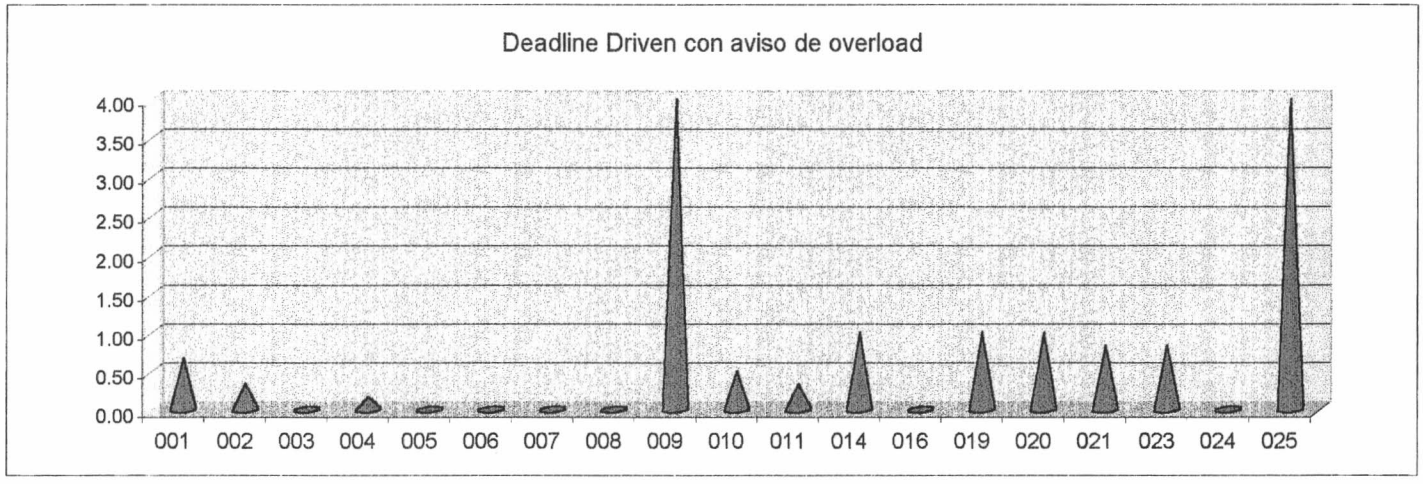
Metas aperiódicas perdidas

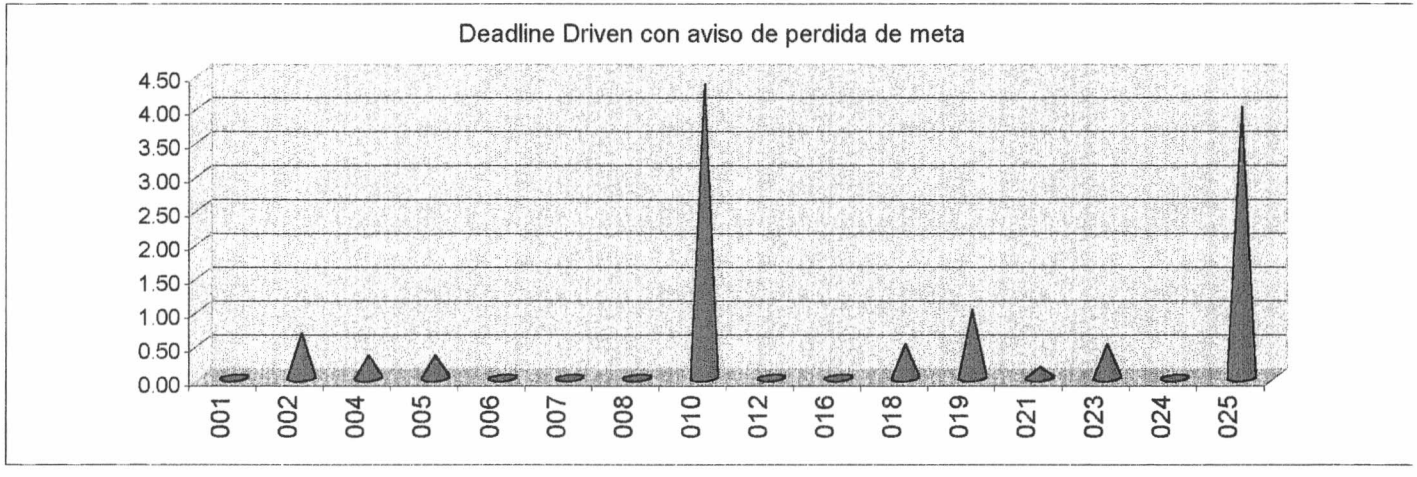


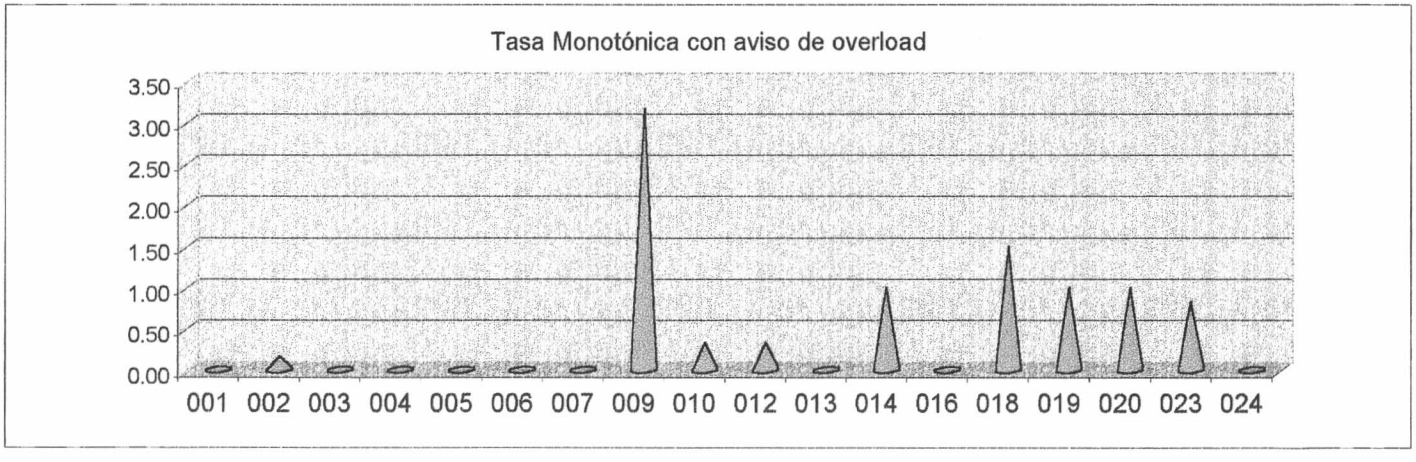




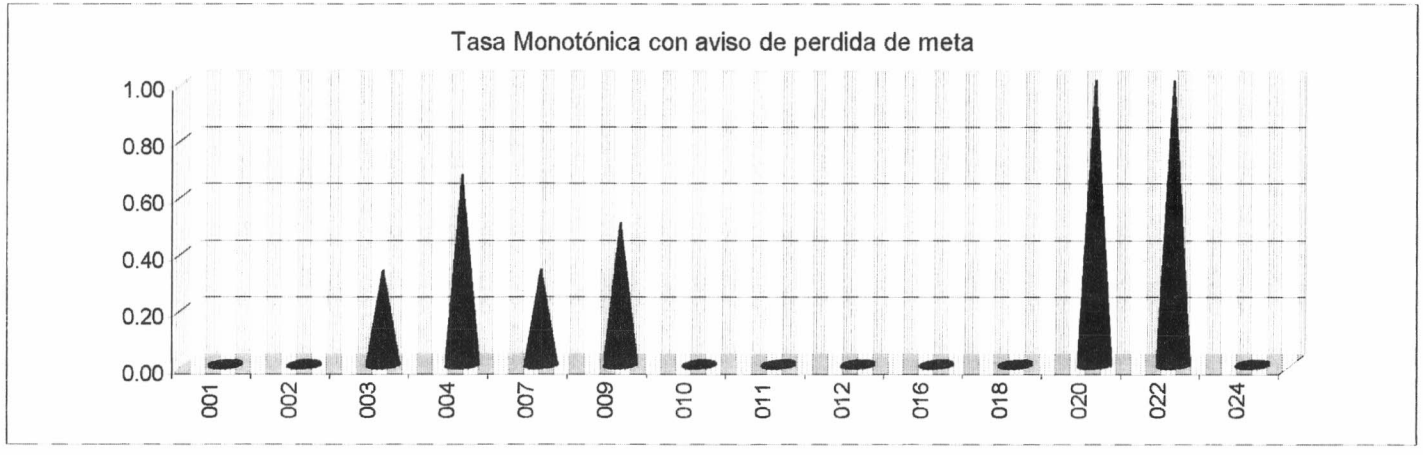
Metas aperiódicas perdidas







Metas aperiódicas perdidas

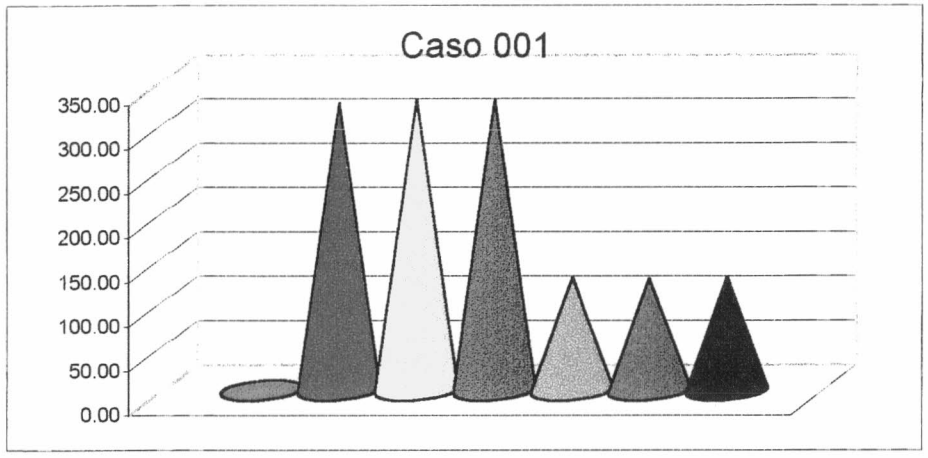


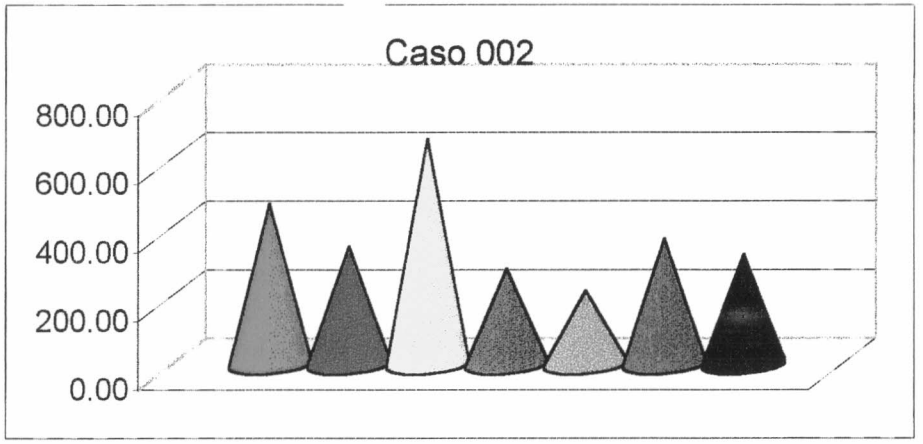
Lotes de Prueba – Cantidad de Metas Periódicas Perdidas

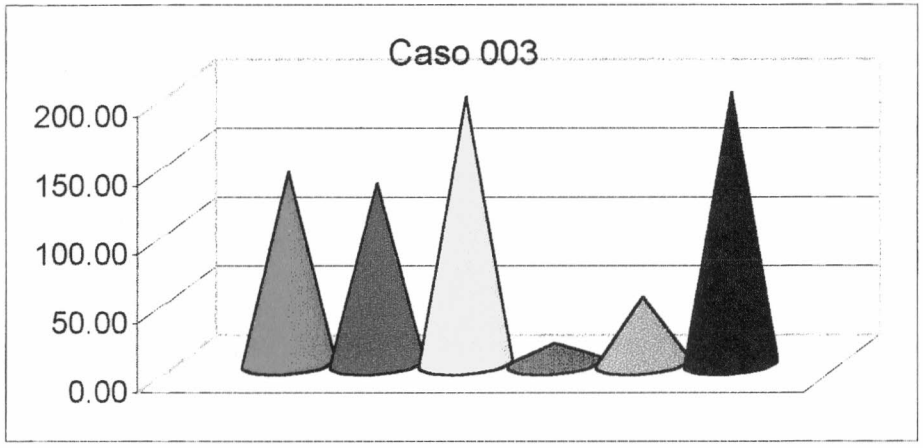
Los siguientes gráficos muestran como se comportó cada caso de prueba con los algoritmos que fueron testeados en relación a la perdida de metas de las tareas periódicas. En este caso la comparación es con el mismo lote de prueba y los distintos algoritmos.

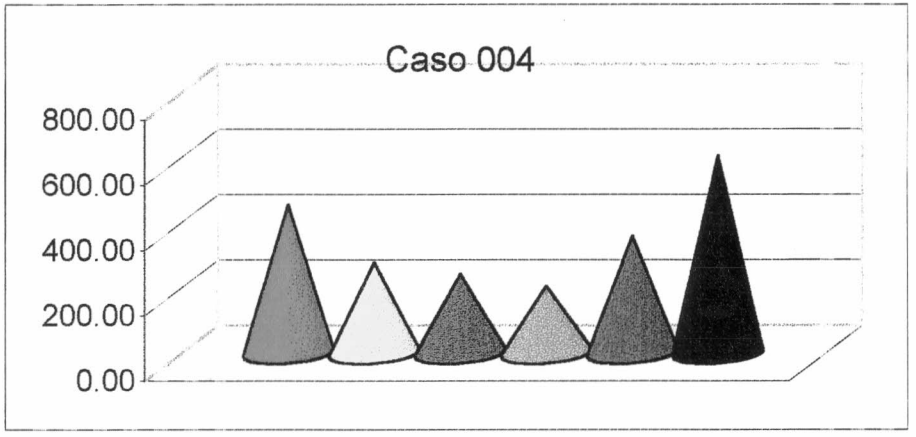
Referencia:
Algoritmos:

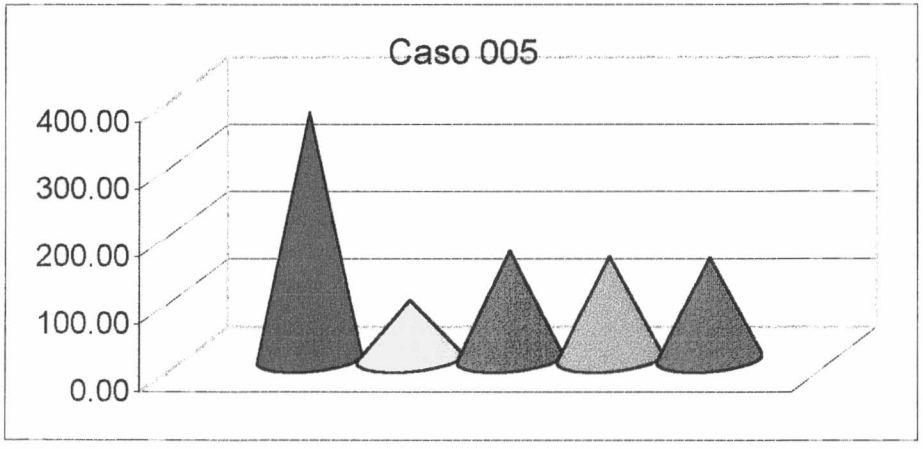
- Deadline Driven
- Tasa Monotónica
- Least Laxity First
- Deadline Driven con Aviso de Overload
- Tasa Monotónica con Aviso de Overload
- Deadline Driven con Aviso de Perdida de Meta
- Tasa Monotónica con Aviso de Perdida de Meta

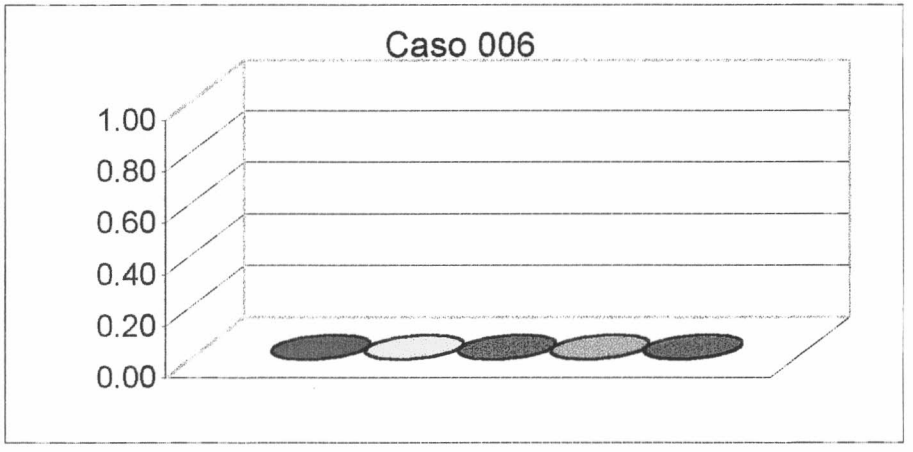

Caso 001
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00

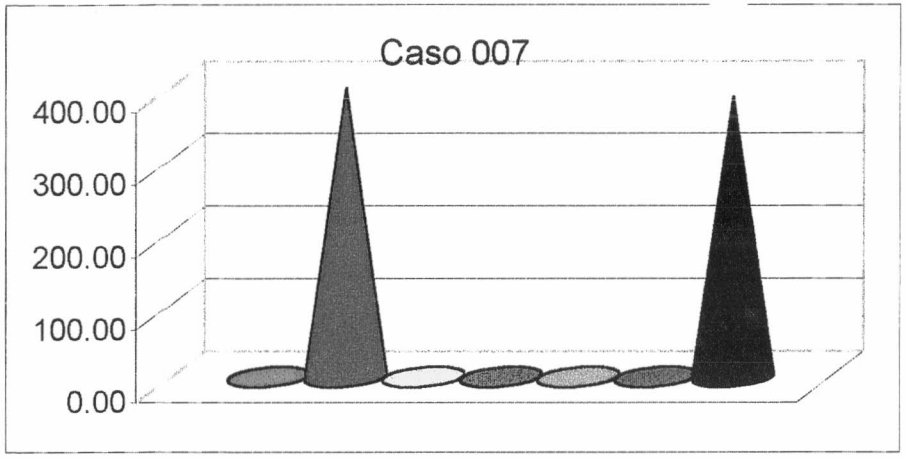

Caso 002
800.00
600.00
400.00
200.00
0.00

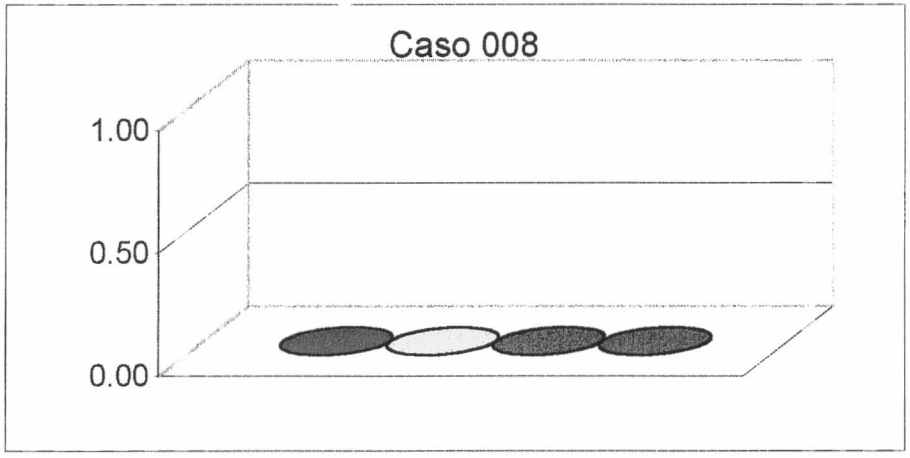

Caso 003
200.00
150.00
100.00
50.00
0.00

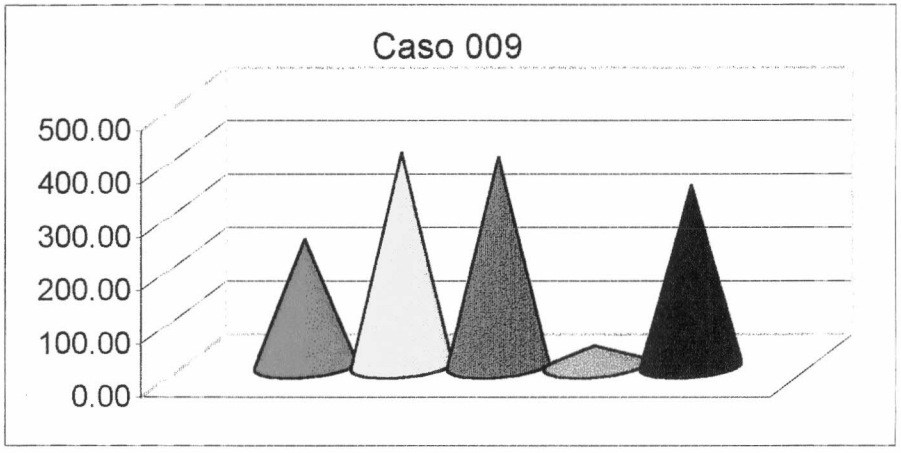

Caso 004
800.00
600.00
400.00
200.00
0.00

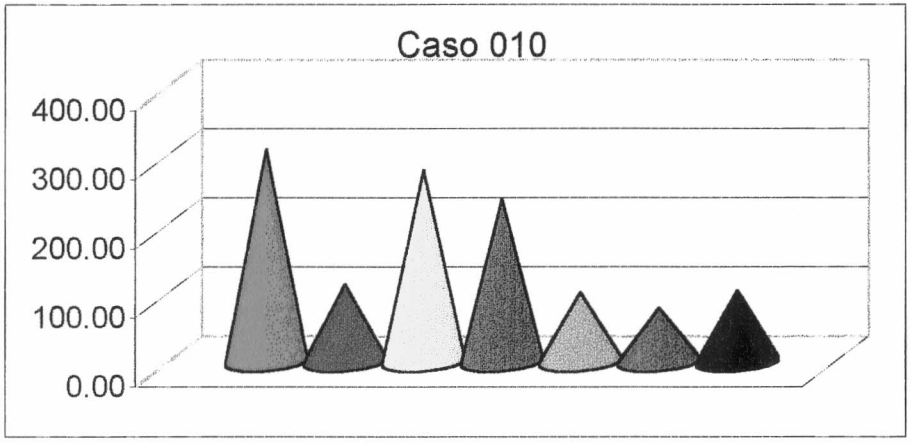

Caso 005
400.00
300.00
200.00
100.00
0.00

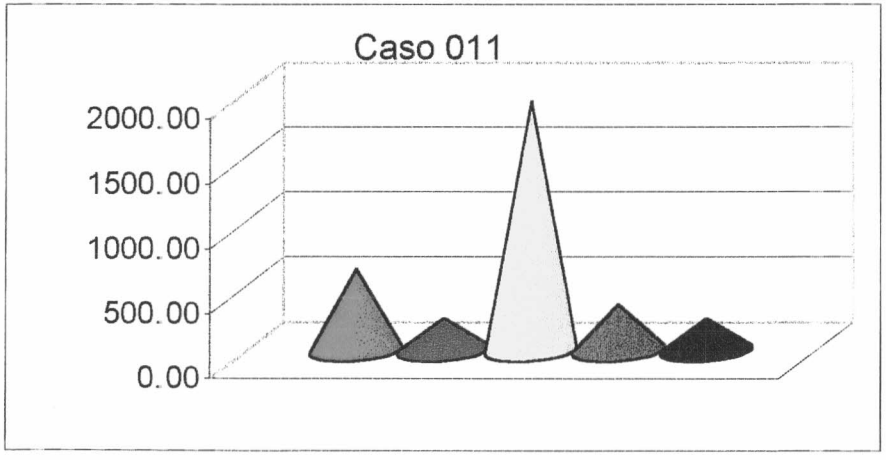

Caso 006
1.00
0.80
0.60
0.40
0.20
0.00

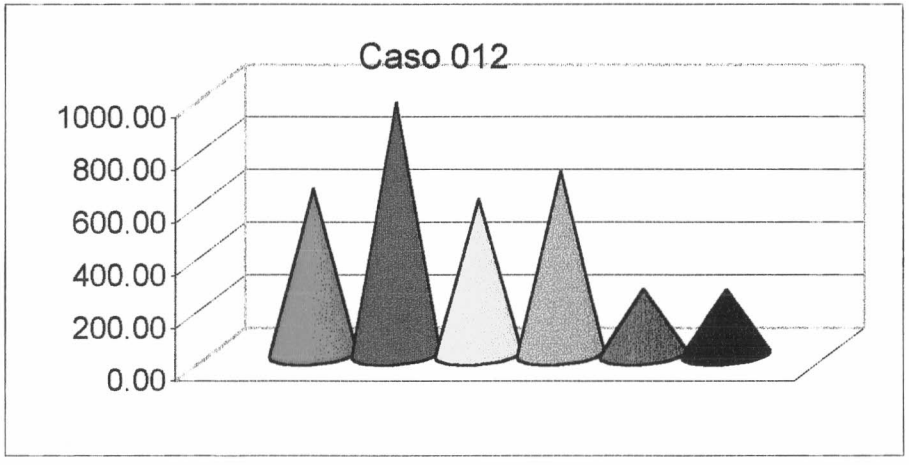

Caso 007
400.00
300.00
200.00
100.00
0.00

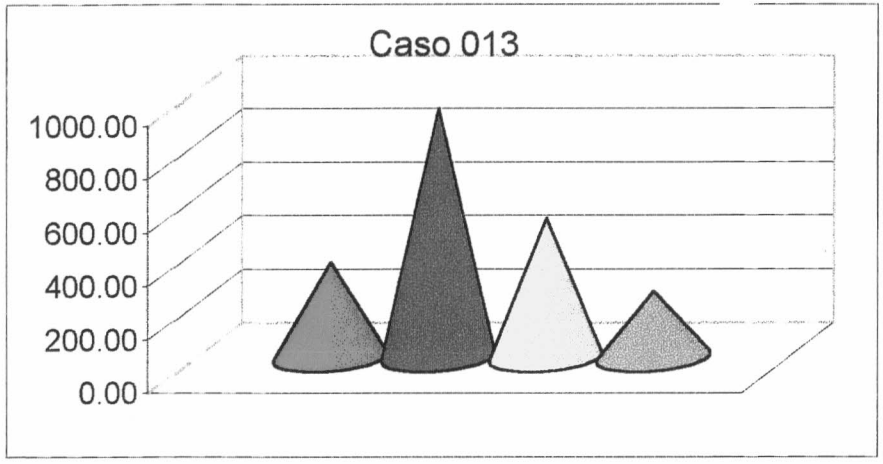

Caso 008
1.00
0.50
0.00

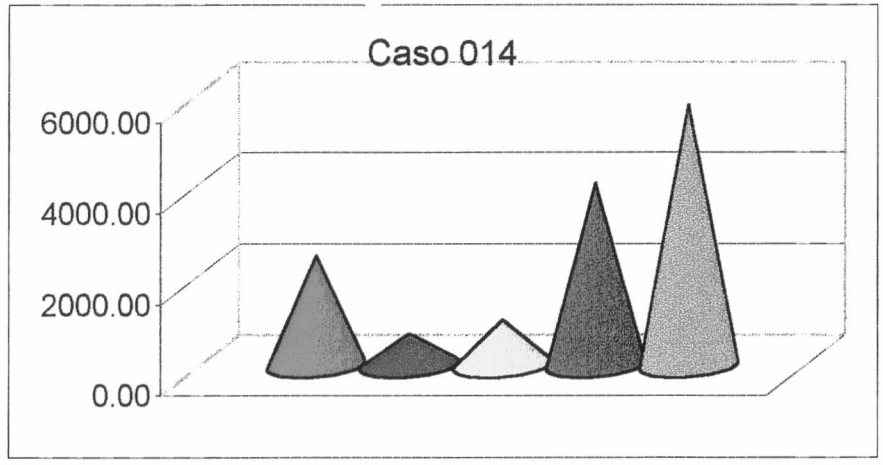

Caso 009
500.00
400.00
300.00
200.00
100.00
0.00

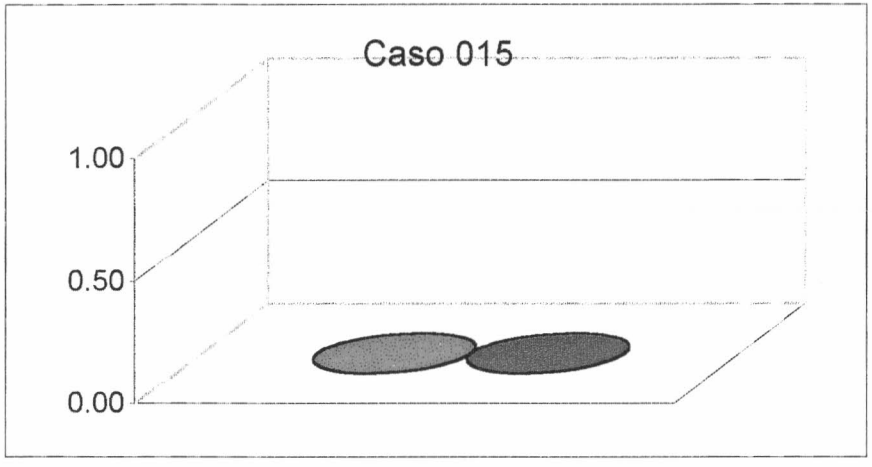

Caso 010
400.00
300.00
200.00
100.00
0.00

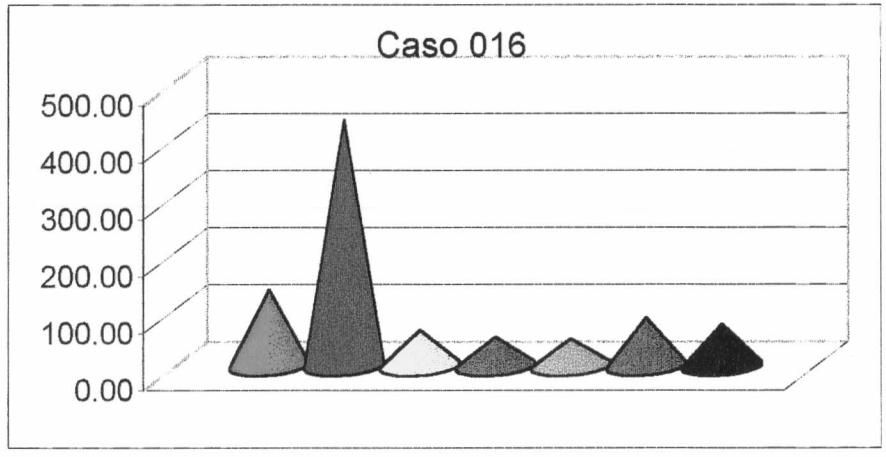

Caso 011
2000.00
1500.00
1000.00
500.00
0.00

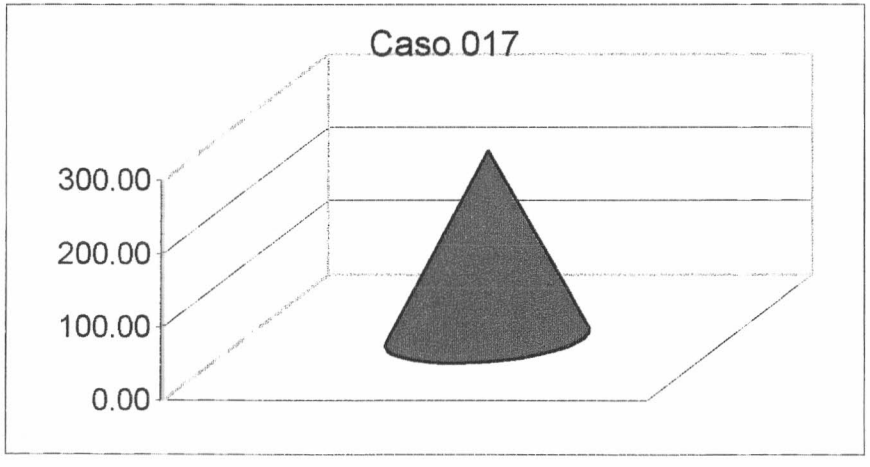

Caso 012
1000.00
800.00
600.00
400.00
200.00
0.00

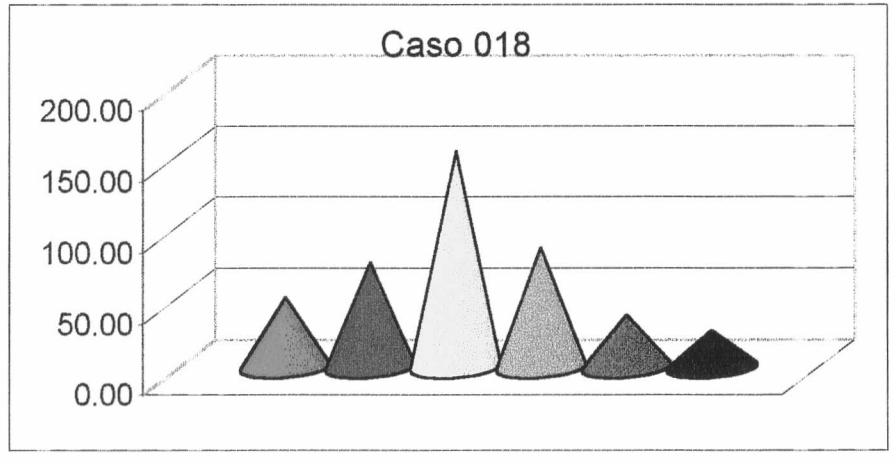

Caso 013
1000.00
800.00
600.00
400.00
200.00
0.00

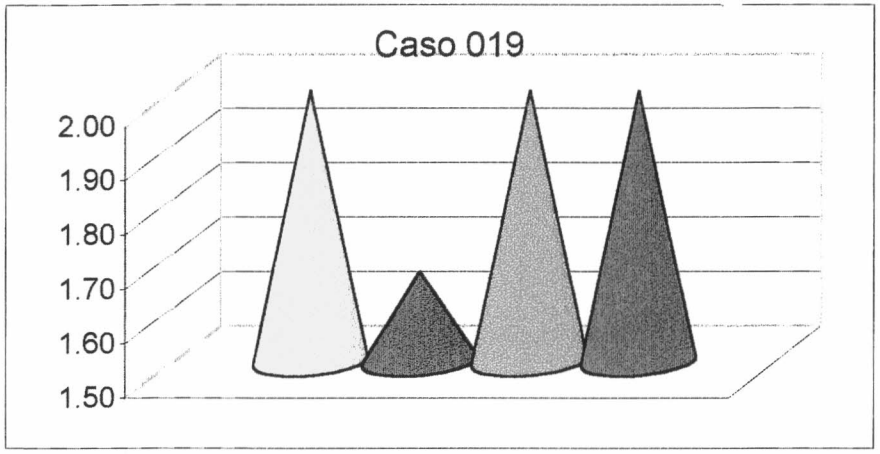

Caso 014
6000.00
4000.00
2000.00
0.00

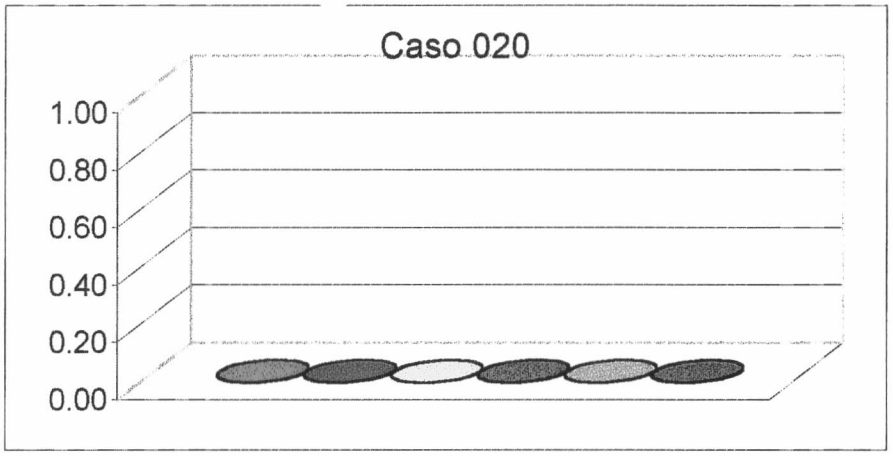

Caso 015
1.00
0.50
0.00

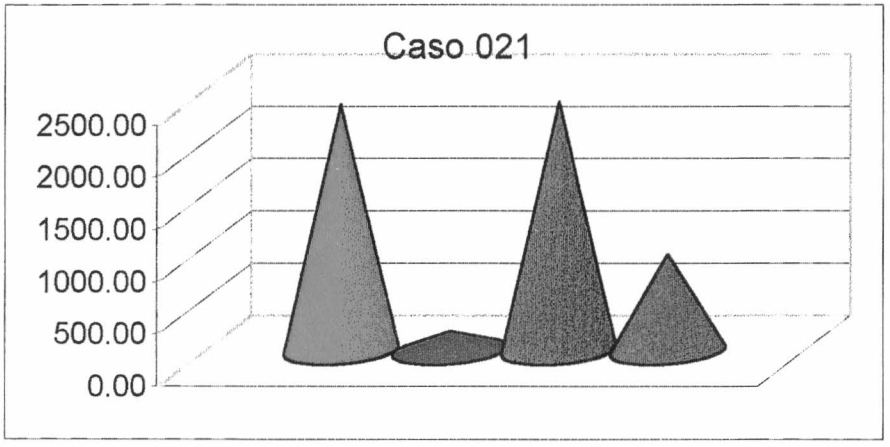

Caso 016
500.00
400.00
300.00
200.00
100.00
0.00

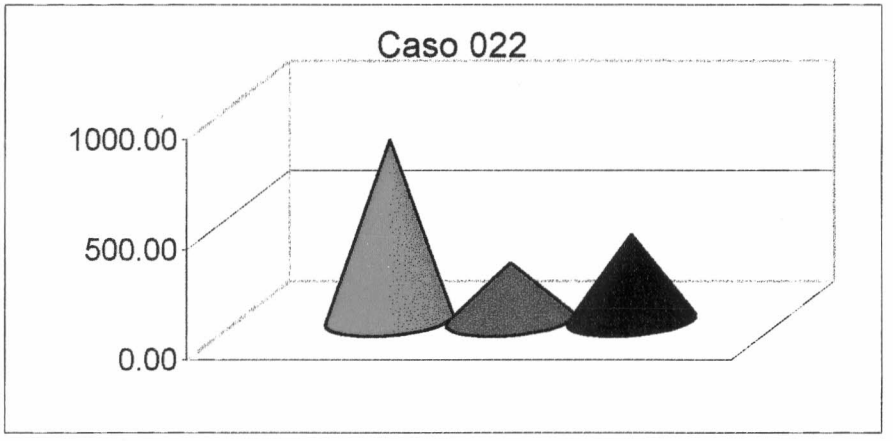

Caso 017
300.00
200.00
100.00
0.00

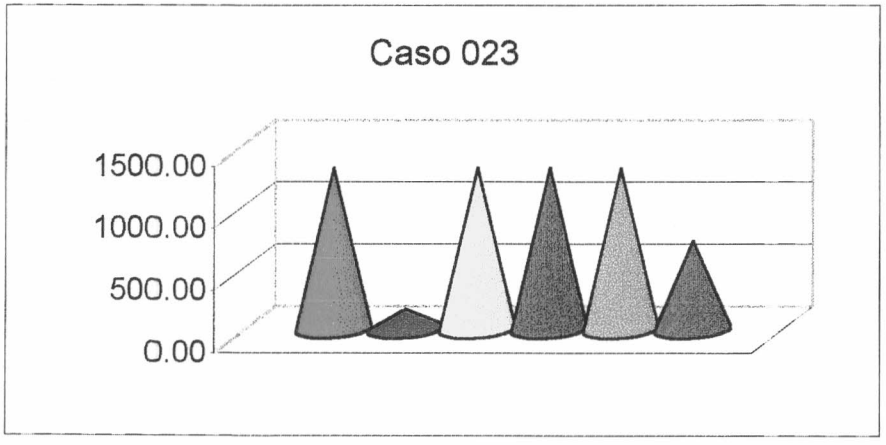

Caso 018
200.00
150.00
100.00
50.00
0.00

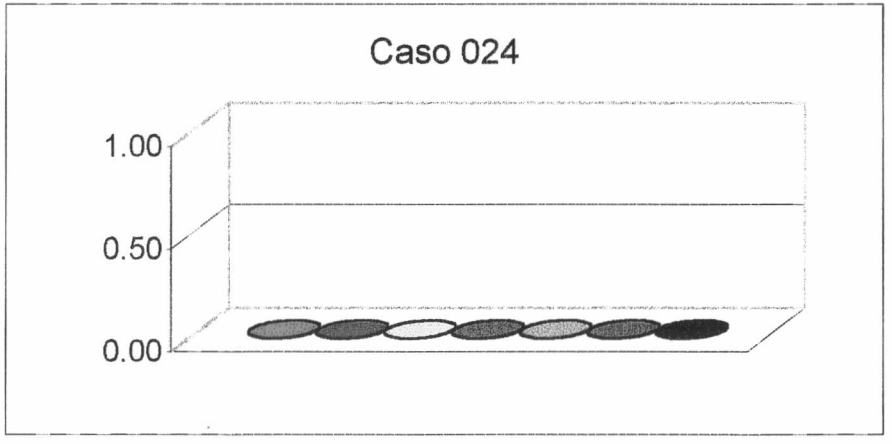

Caso 019
2.00
1.90
1.80
1.70
1.60
1.50

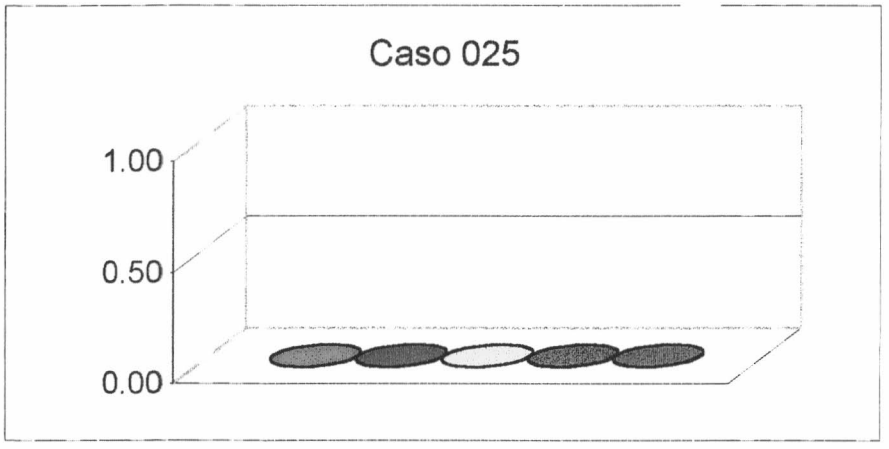

Caso 020
1.00
0.80
0.60
0.40
0.20
0.00

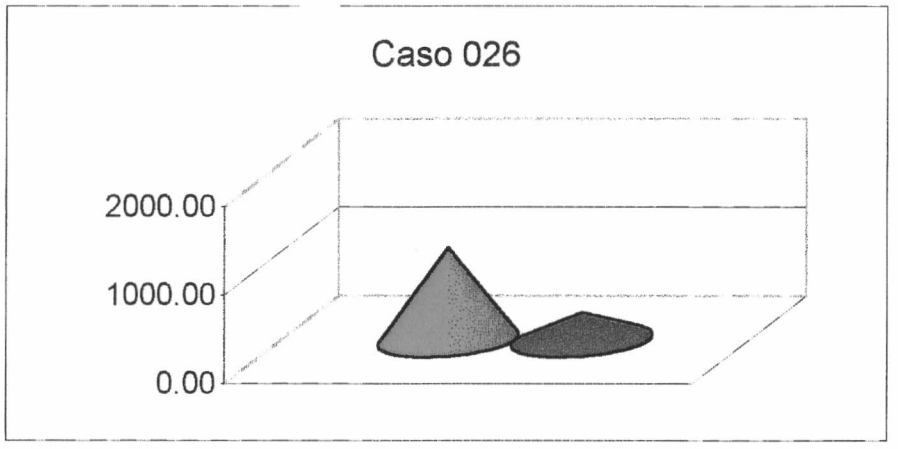

Caso 021
2500.00
2000.00
1500.00
1000.00
500.00
0.00


Caso 022
1000.00
500.00
0.00


Caso 023
1500.00
1000.00
500.00
0.00


Caso 024
1.00
0.50
0.00


Caso 025
1.00
0.50
0.00

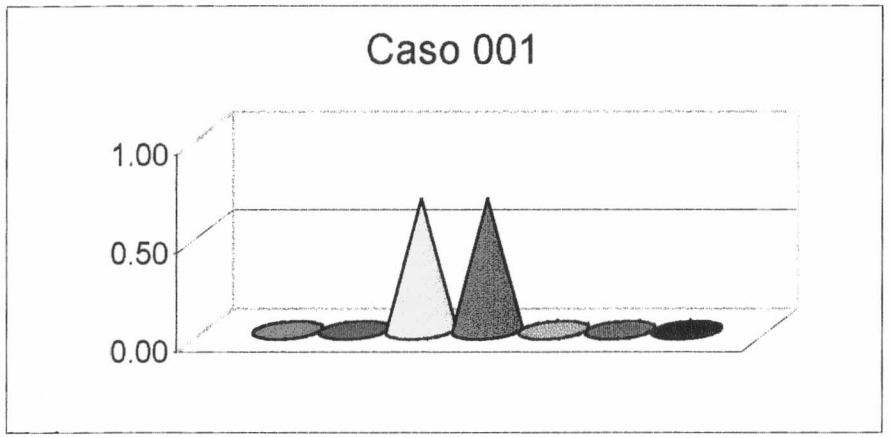

Caso 026
2000.00
1000.00
0.00

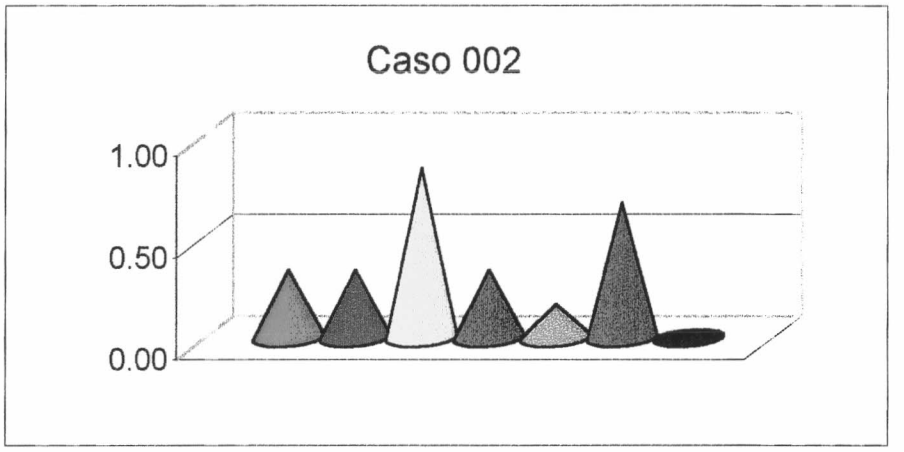
Lotes de Prueba – Cantidad de Metas Aperiódicas Perdidas

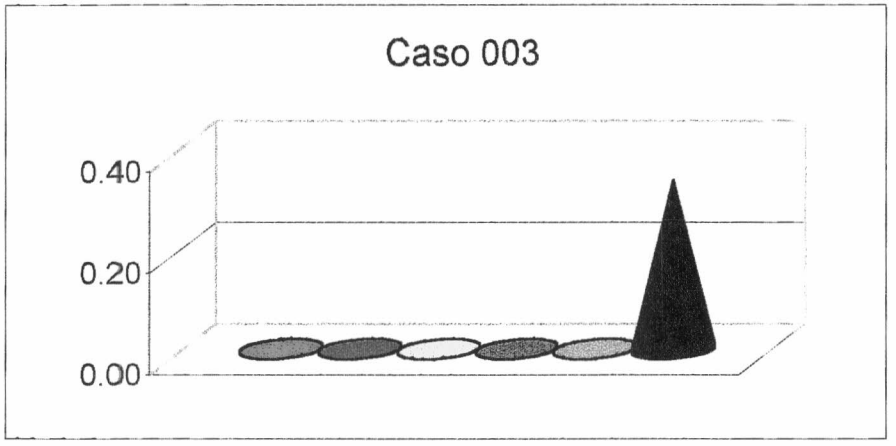
Los siguientes gráficos muestran como se comportó cada caso de prueba con los algoritmos que fueron testeados en relación a la perdida de metas de las tareas aperiódicas.

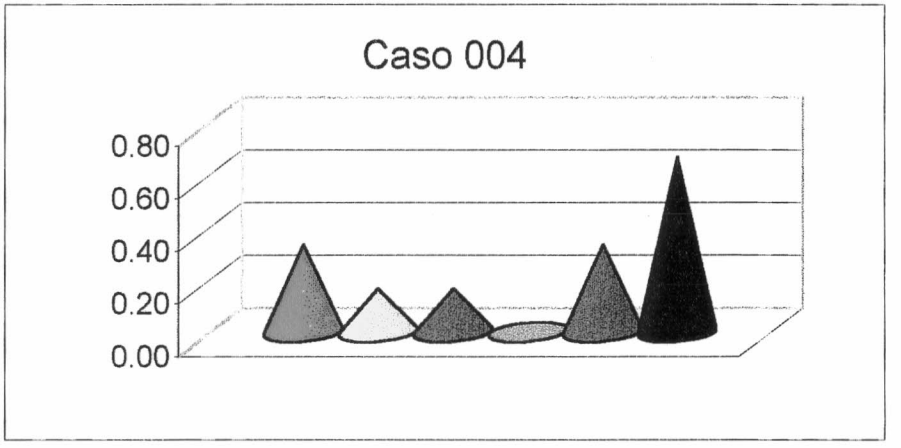
Referencia:
Algoritmos:

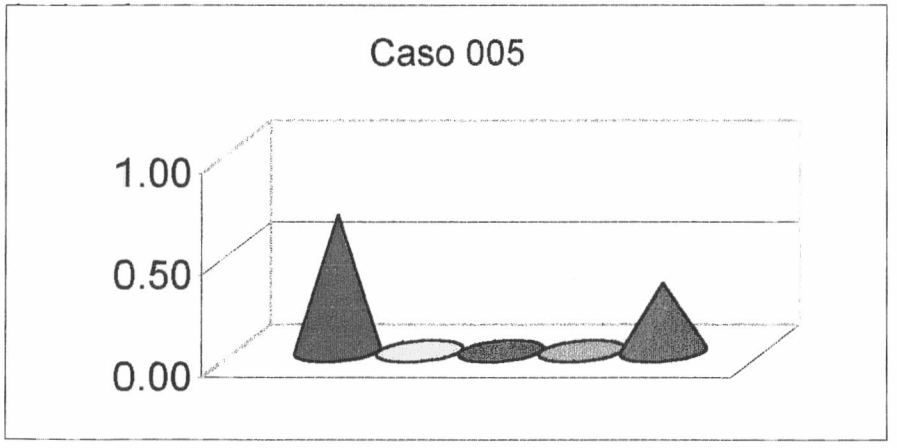
- Deadline Driven
- Tasa Monotónica
- Least Laxity First
- Deadline Driven con Aviso de Overload
- Tasa Monotónica con Aviso de Overload
- Deadline Driven con Aviso de Perdida de Meta
- Tasa Monotónica con Aviso de Perdida de Meta

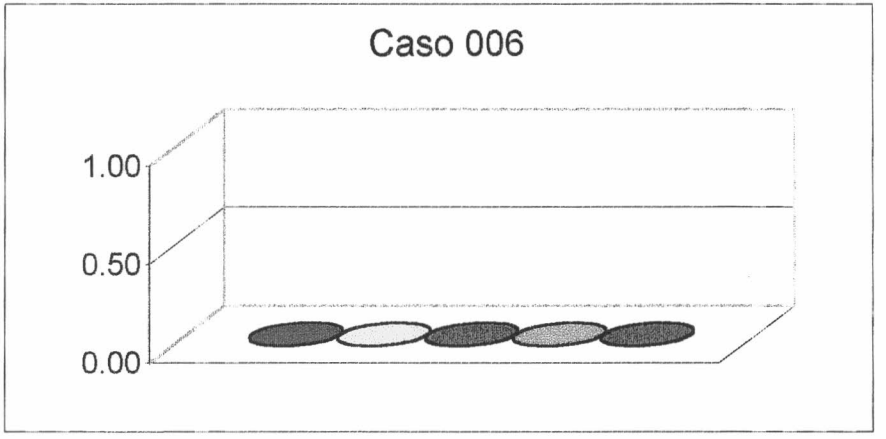

Caso 001
1.00
0.50
0.00

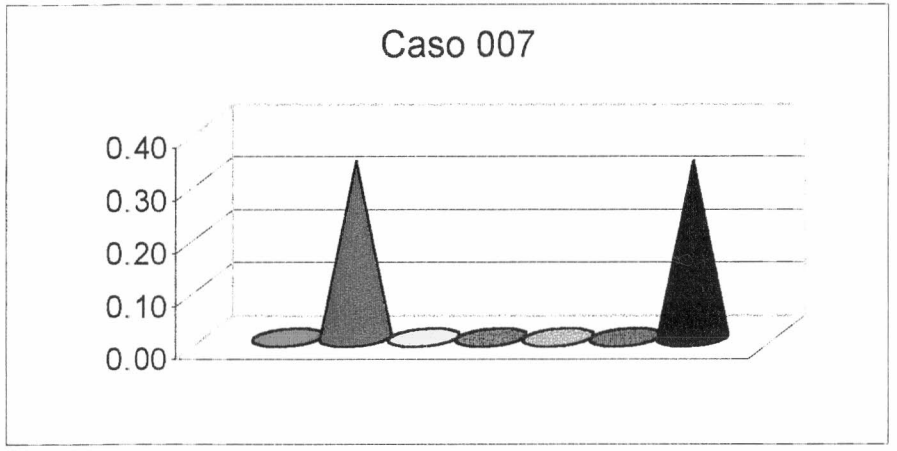

Caso 002
1.00
0.50
0.00

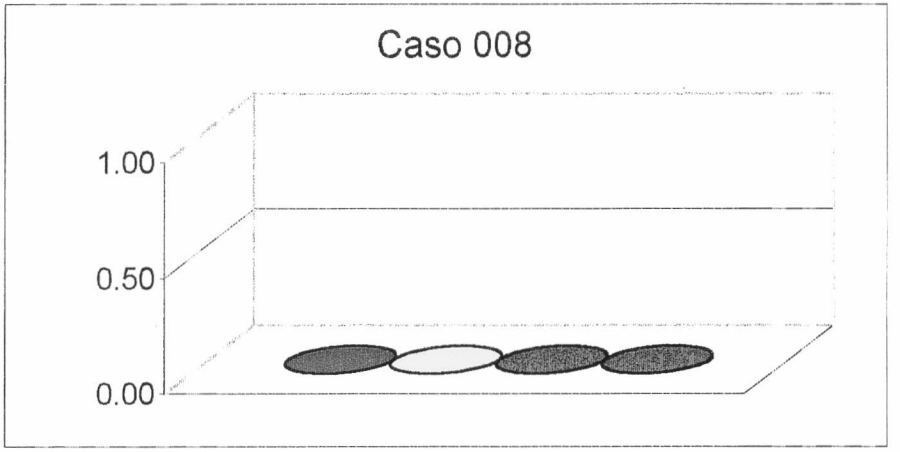

Caso 003
0.40
0.20
0.00

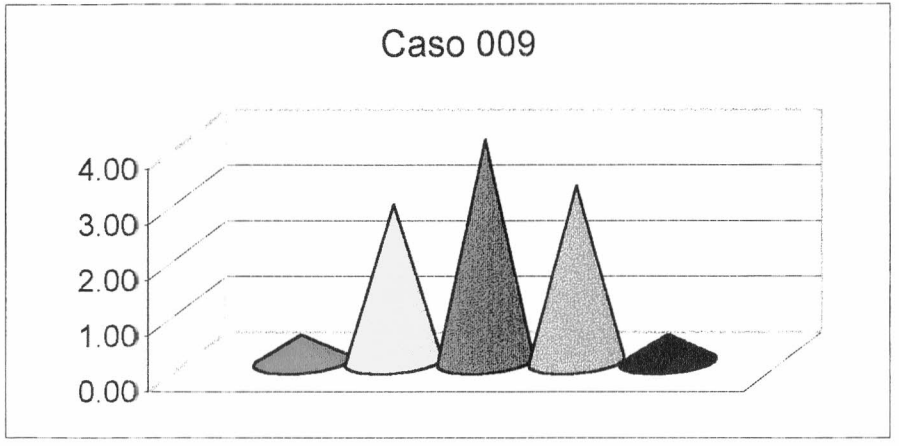

Caso 004
0.80
0.60
0.40
0.20
0.00

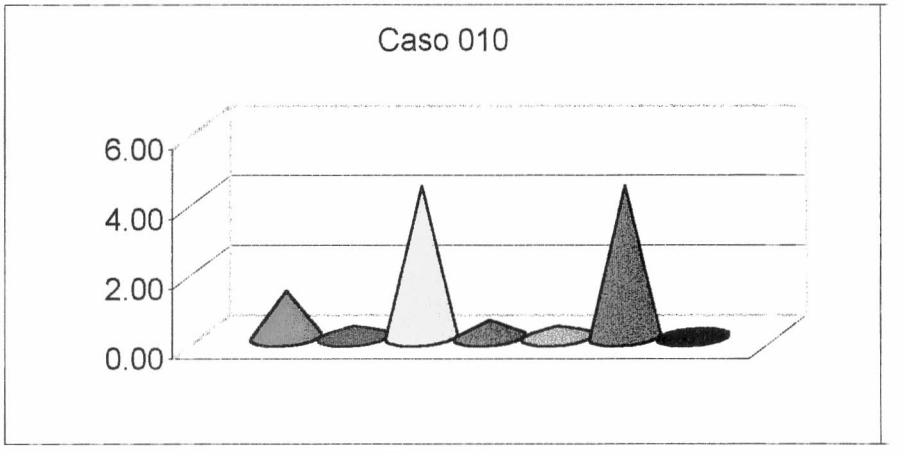

Caso 005
1.00
0.50
0.00

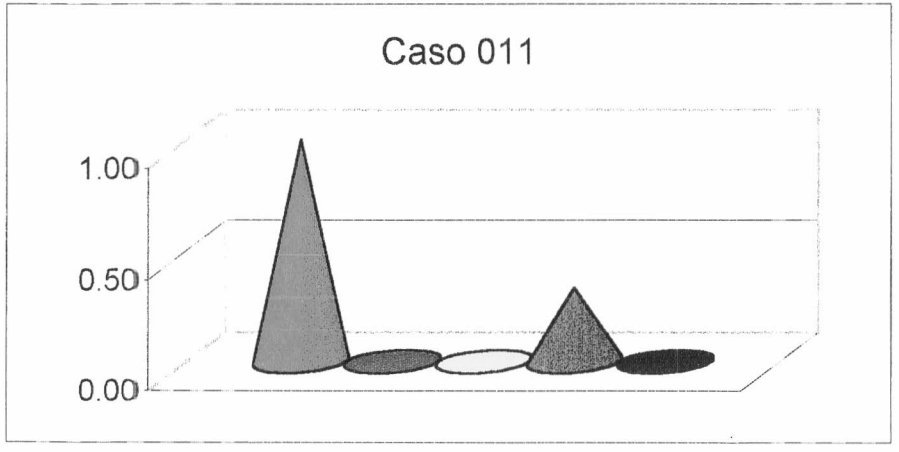

Caso 006
1.00
0.50
0.00

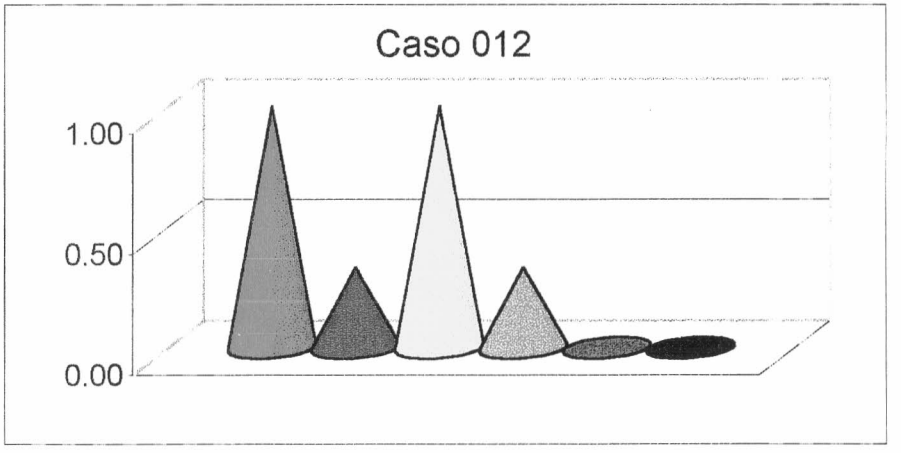

Caso 007
0.40
0.30
0.20
0.10
0.00

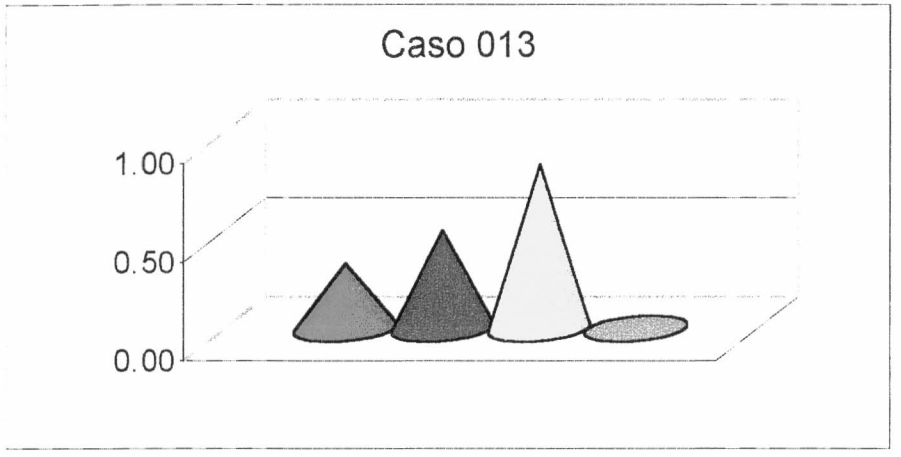

Caso 008
1.00
0.50
0.00

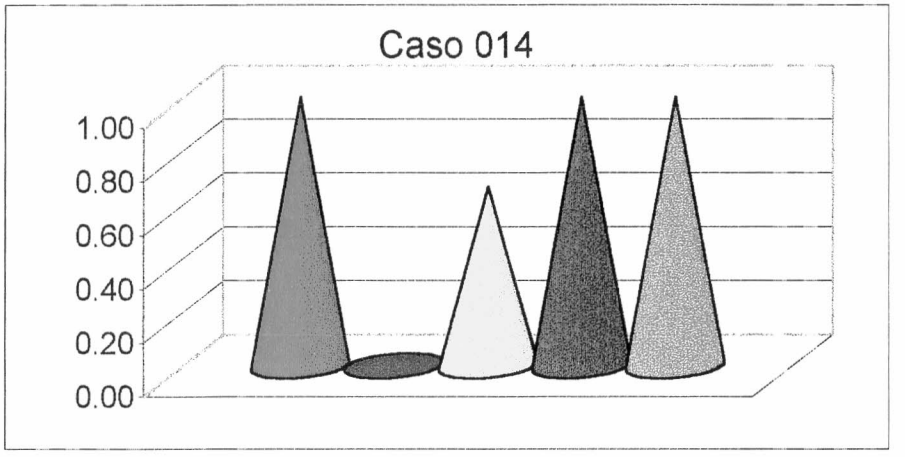

Caso 009
4.00
3.00
2.00
1.00
0.00

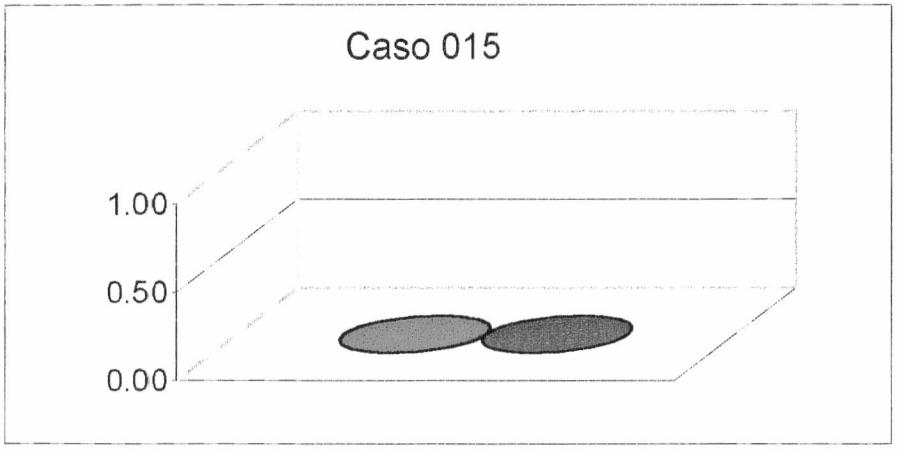

Caso 010
6.00
4.00
2.00
0.00

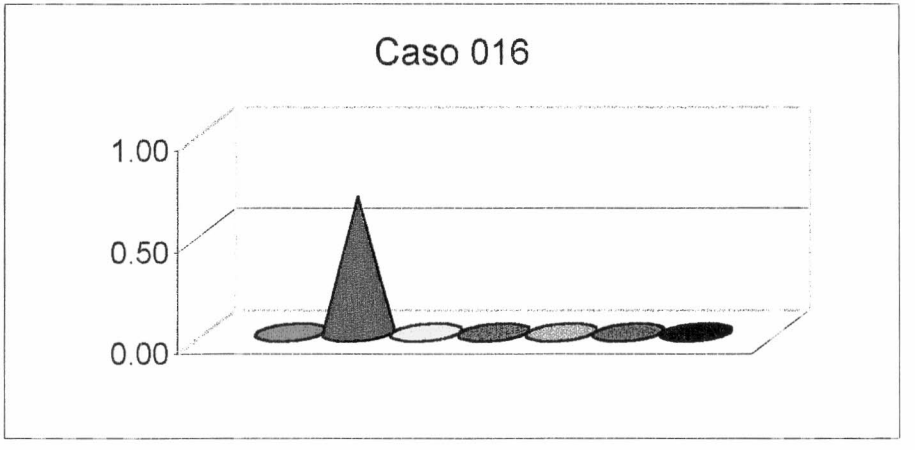

Caso 011
1.00
0.50
0.00

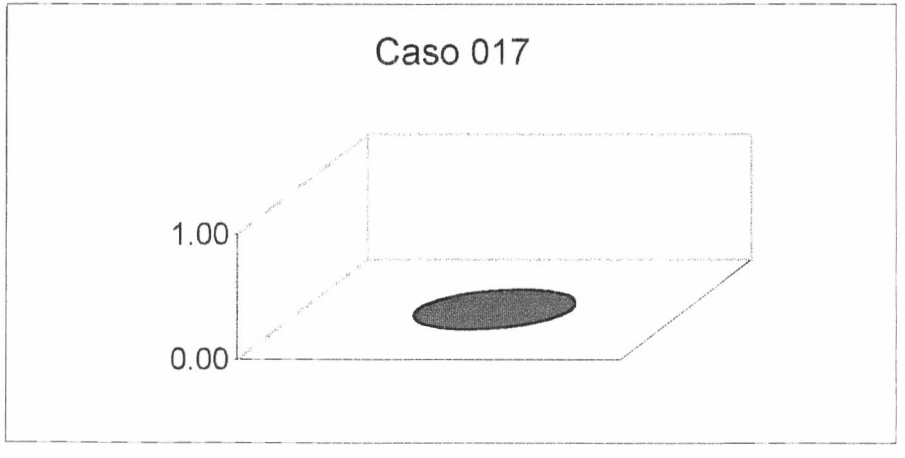

Caso 012
1.00
0.50
0.00

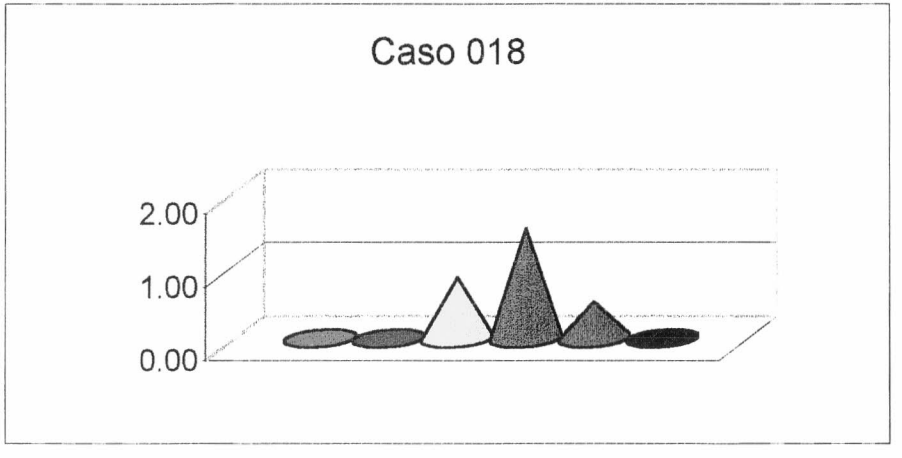

Caso 013
1.00
0.50
0.00

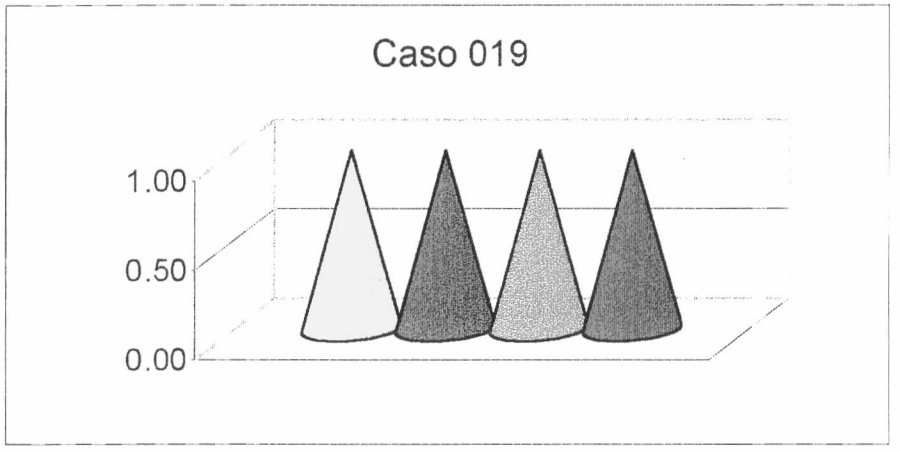

Caso 014
1.00
0.80
0.60
0.40
0.20
0.00

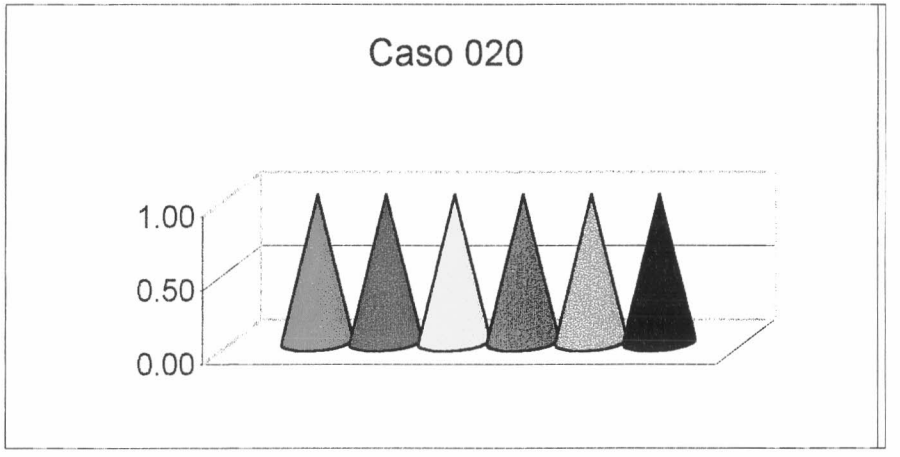

Caso 015
1.00
0.50
0.00

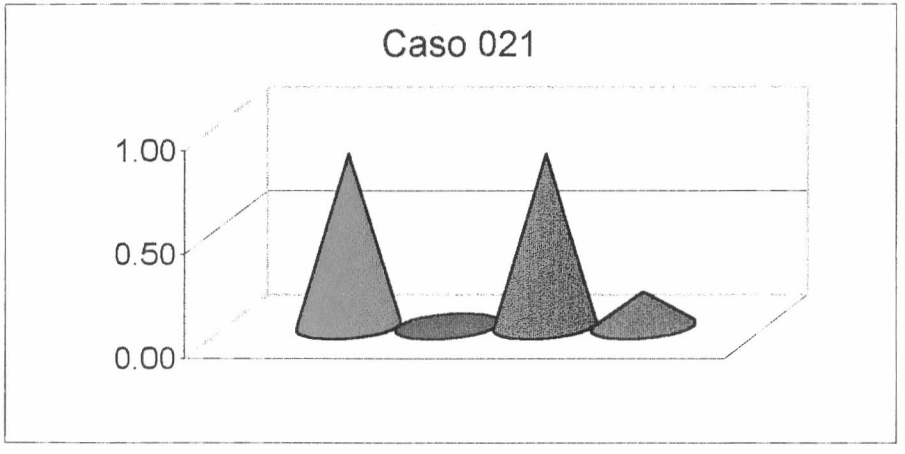

Caso 016
1.00
0.50
0.00

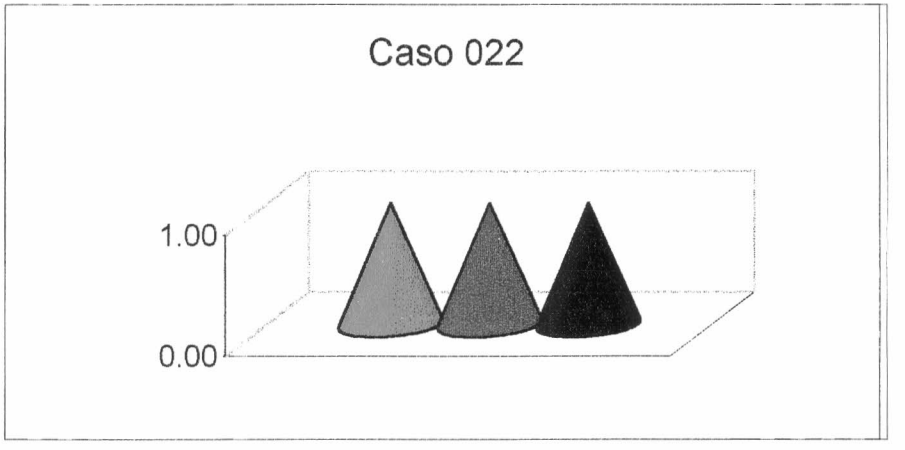

Caso 017
1.00
0.00

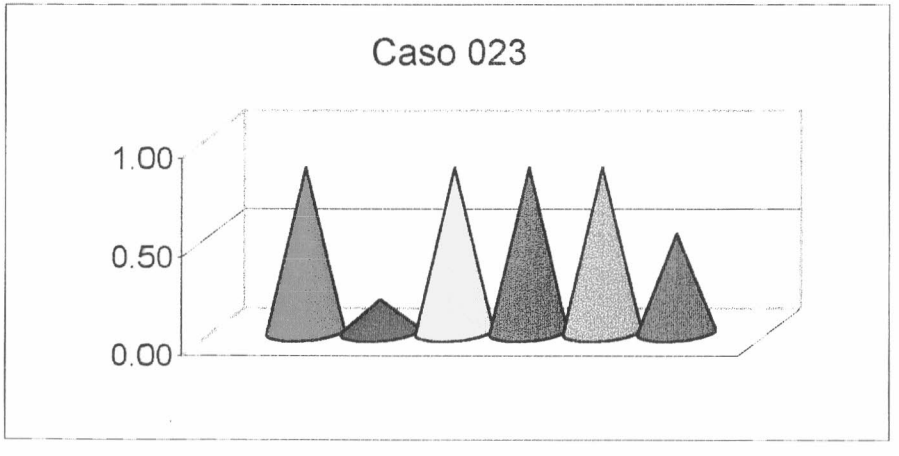

Caso 018
2.00
1.00
0.00

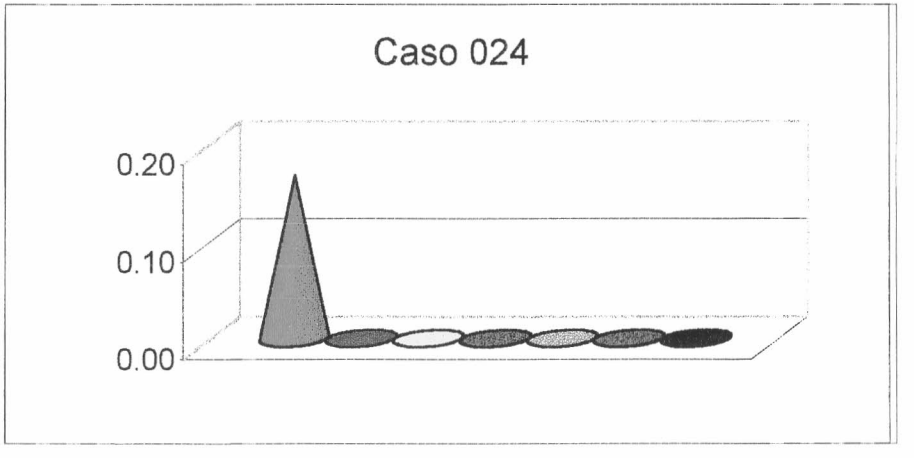

Caso 019
1.00
0.50
0.00

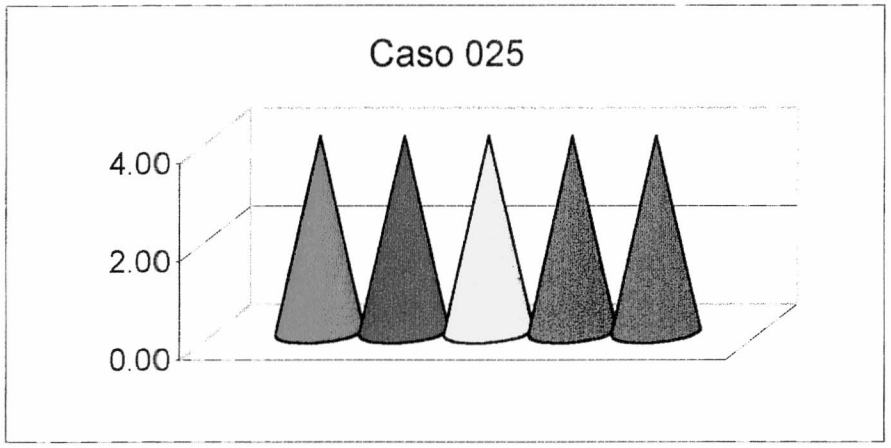

Caso 020
1.00
0.50
0.00

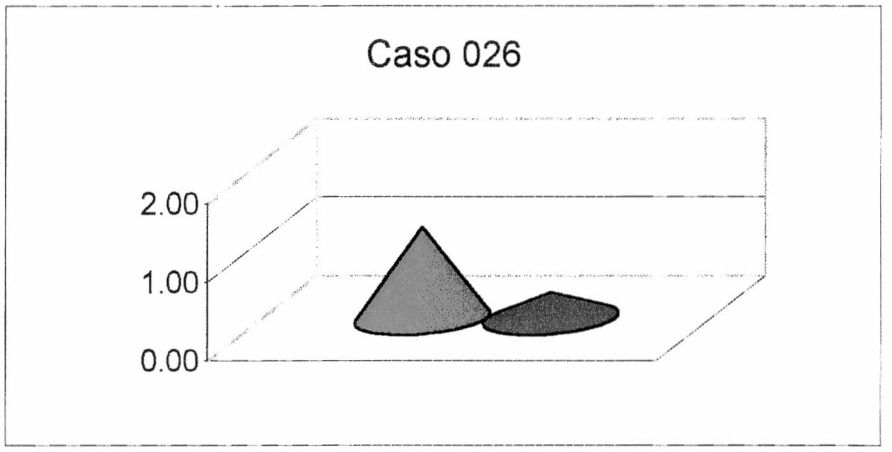

Caso 021
1.00
0.50
0.00


Caso 022
1.00
0.00


Caso 023
1.00
0.50
0.00

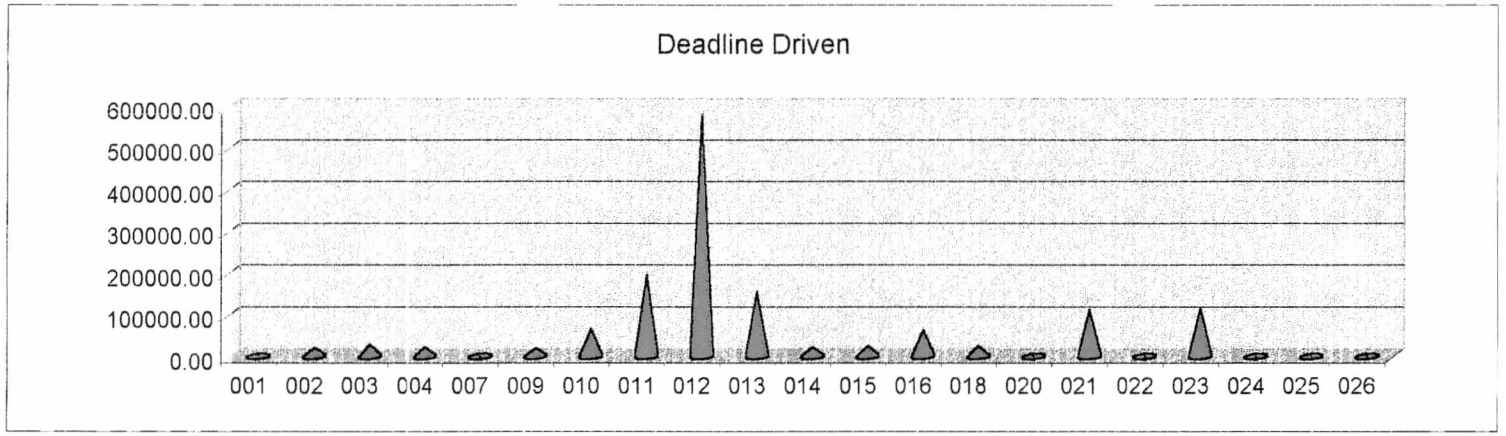

Caso 024
0.20
0.10
0.00

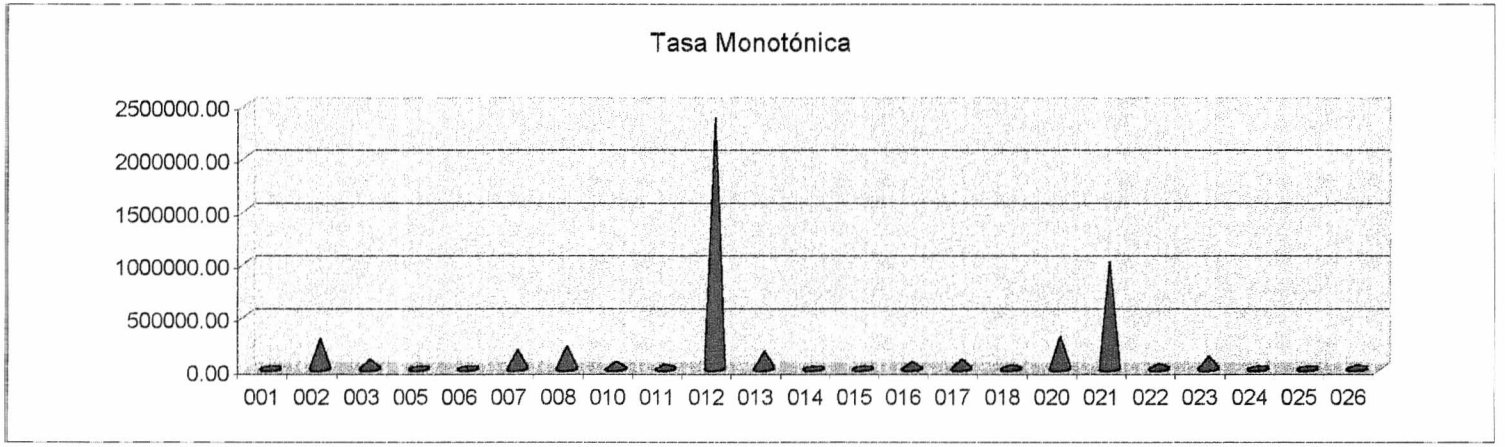

Caso 025
4.00
2.00
0.00

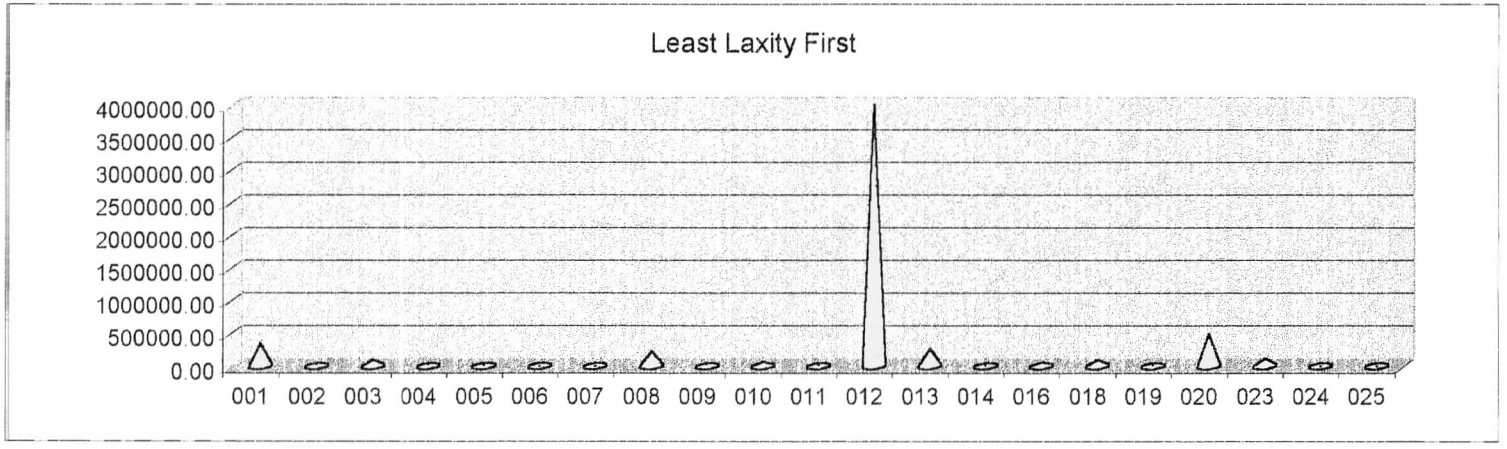

Caso 026
2.00
1.00
0.00

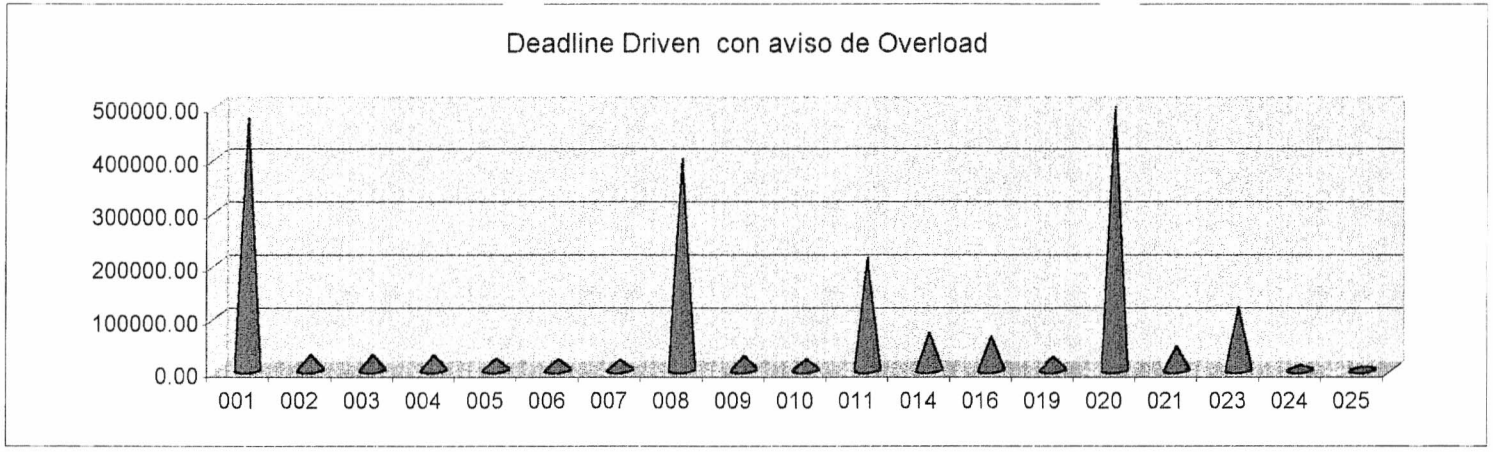
Algoritmos – Tiempo Ocioso que tiene el procesador

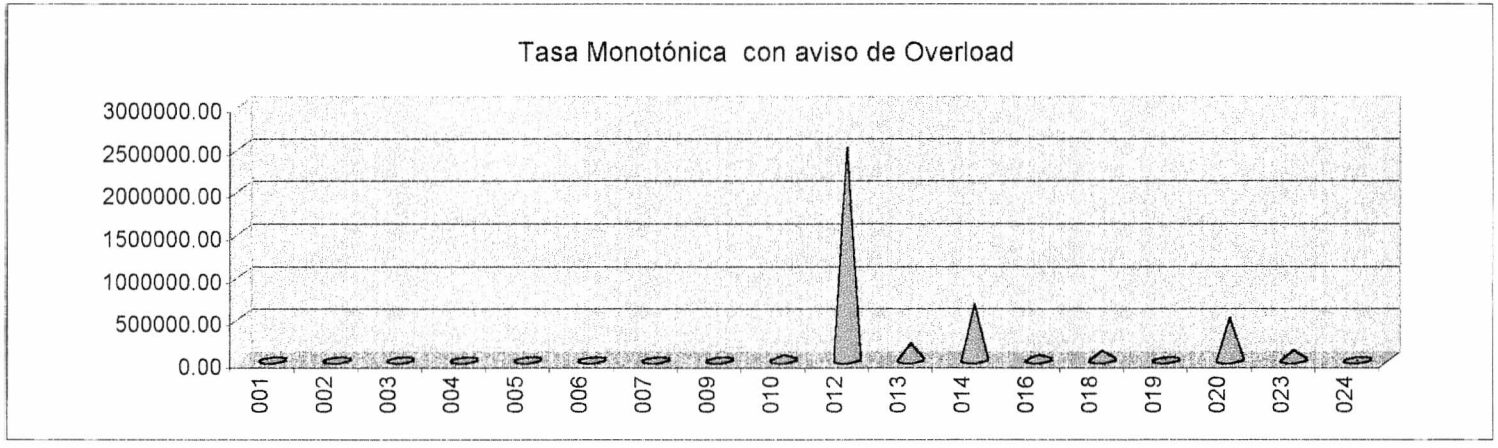
En los siguientes gráficos se muestra una comparación del tiempo ocioso del procesador durante la ejecución de los distintos algoritmos testeados.

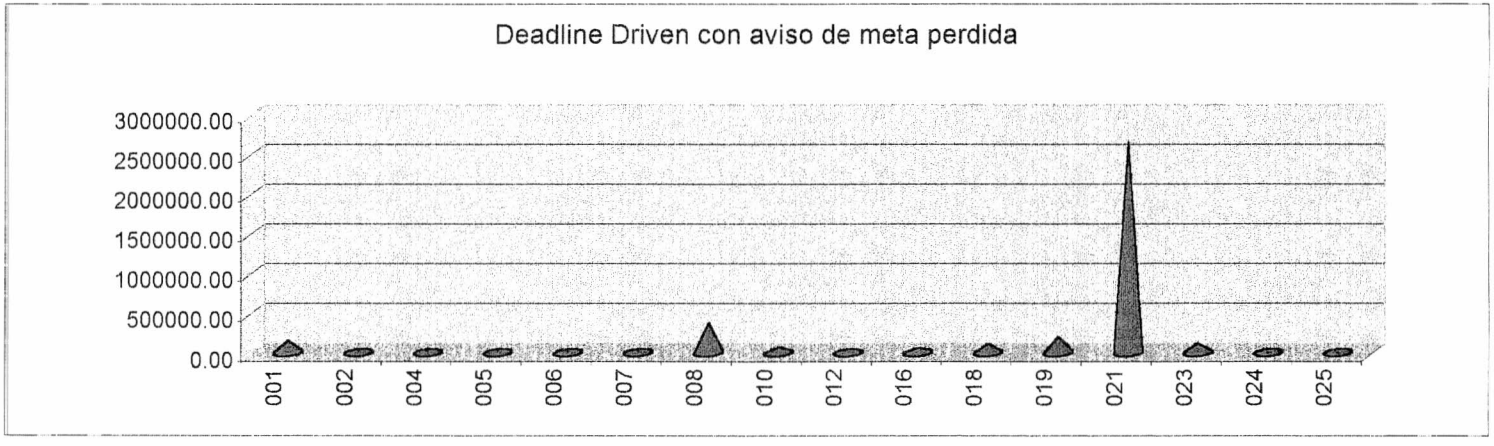

Deadline Driven
600000.00
500000.00
400000.00
300000.00
200000.00
100000.00
0.00
001 002 003 004 007 009 010 011 012 013 014 015 016 018 020 021 022 023 024 025 026

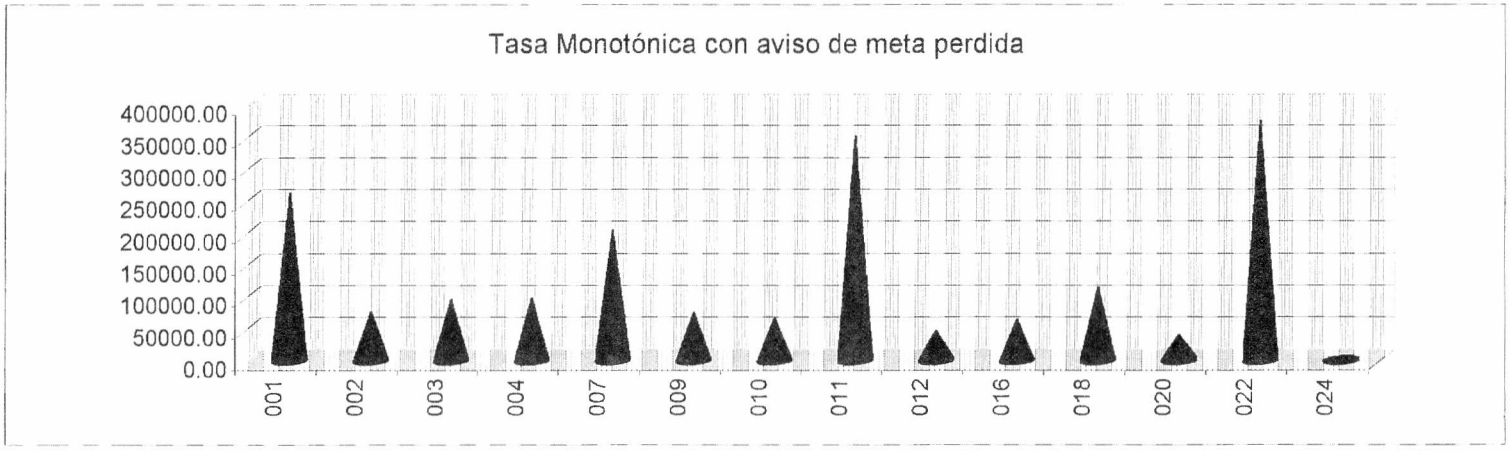

Tasa Monotónica
2500000.00
2000000.00
1500000.00
1000000.00
500000.00
0.00
001 002 003 005 006 007 008 010 011 012 013 014 015 016 017 018 020 021 022 023 024 025 026


Least Laxity First
4000000.00
3500000.00
3000000.00
2500000.00
2000000.00
1500000.00
1000000.00
500000.00
0.00
001 002 003 004 005 006 007 008 009 010 011 012 013 014 016 018 019 020 023 024 025


Deadline Driven con aviso de Overload
500000.00
400000.00
300000.00
200000.00
100000.00
0.00
001 002 003 004 005 006 007 008 009 010 011 014 016 019 020 021 023 024 025


Tasa Monotónica con aviso de Overload
3000000.00
2500000.00
2000000.00
1500000.00
1000000.00
500000.00
0.00
001 002 003 004 005 006 007 009 010 012 013 014 016 018 019 020 023 024


Deadline Driven con aviso de meta perdida
3000000.00
2500000.00
2000000.00
1500000.00
1000000.00
500000.00
0.00
001 002 004 005 006 007 008 010 012 016 018 019 021 023 024 025

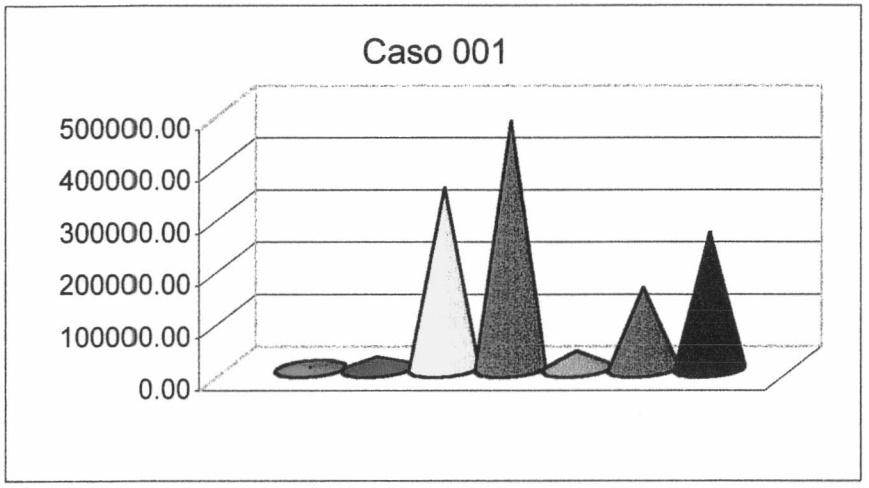

Tasa Monotónica con aviso de meta perdida
400000.00
350000.00
300000.00
250000.00
200000.00
150000.00
100000.00
50000.00
0.00
001
002
003
004
007
009
010
011
012
016
018
020
022
024

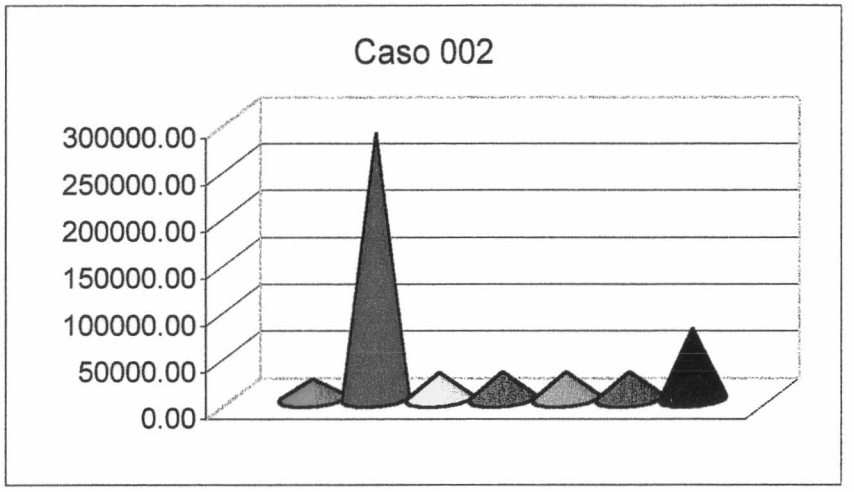
Lotes de Prueba - Tiempo Ocioso empleado por el Procesador

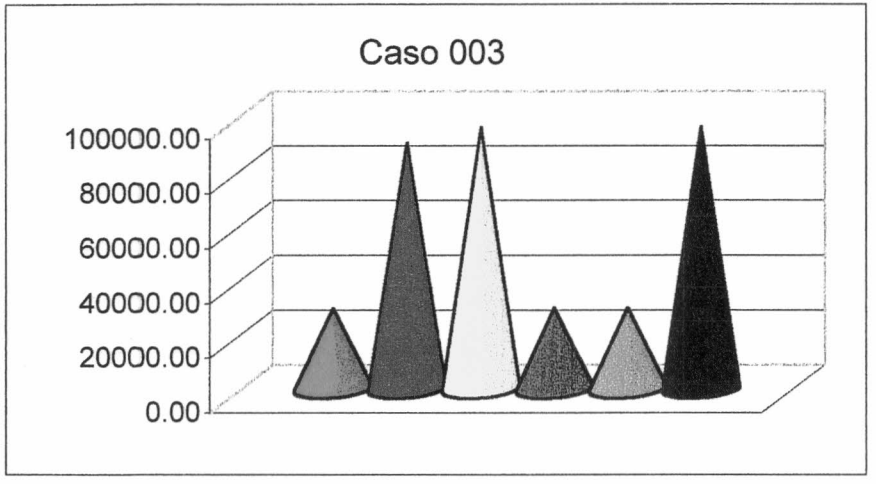
Los siguientes gráficos muestran cuanto tiempo ocioso le quedo al procesador en cada caso de prueba con los algoritmos que fueron testeados.

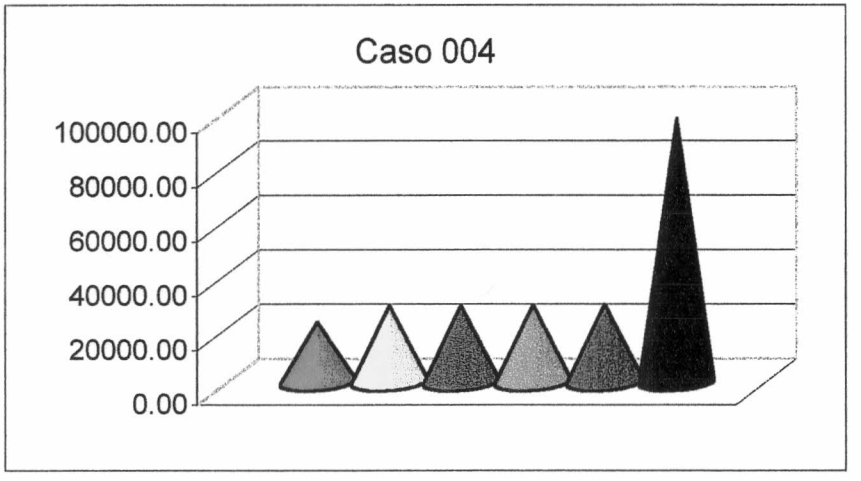
Referencia:
Algoritmos:

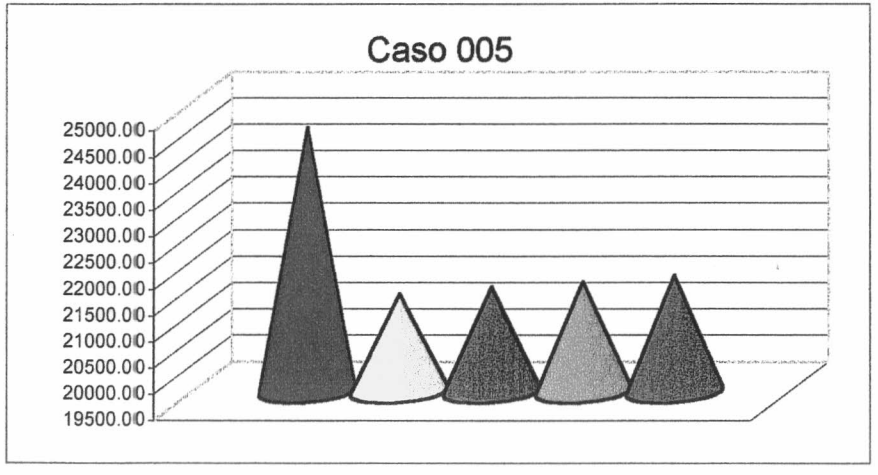
- Deadline Driven
- Tasa Monotónica
- Least Laxity First
- Deadline Driven con Aviso de Overload
- Tasa Monotónica con Aviso de Overload
- Deadline Driven con Aviso de Perdida de Meta
- Tasa Monotónica con Aviso de Perdida de Meta

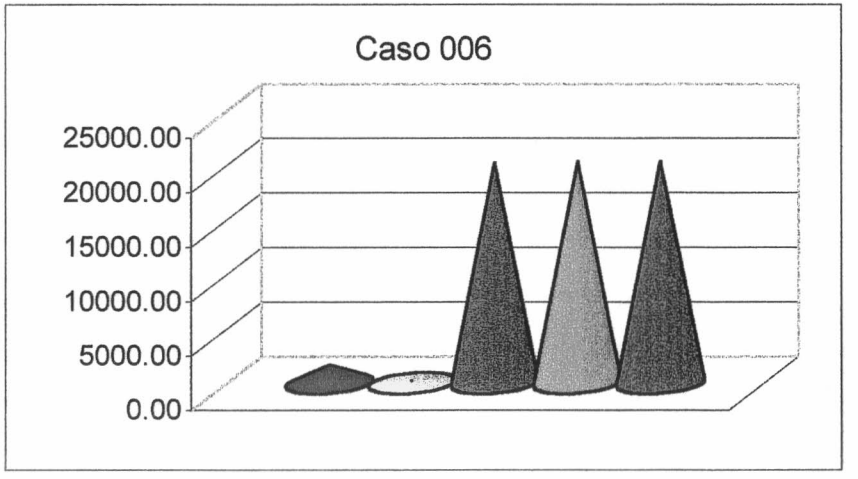

Caso 001
500000.00
400000.00
300000.00
200000.00
100000.00
0.00

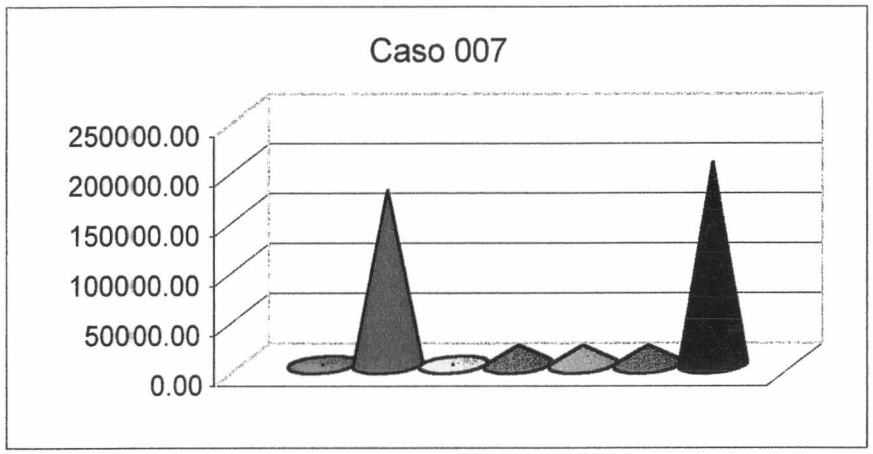

Caso 002
300000.00
250000.00
200000.00
150000.00
100000.00
50000.00
0.00

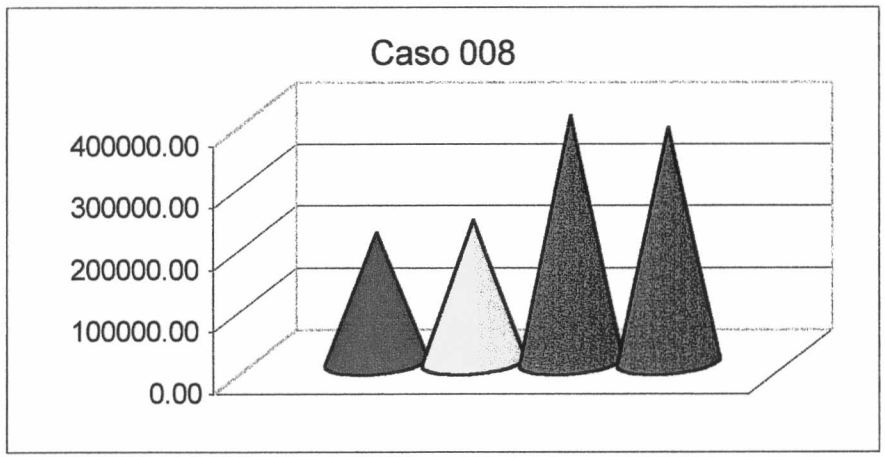

Caso 003
100000.00
80000.00
60000.00
40000.00
20000.00
0.00

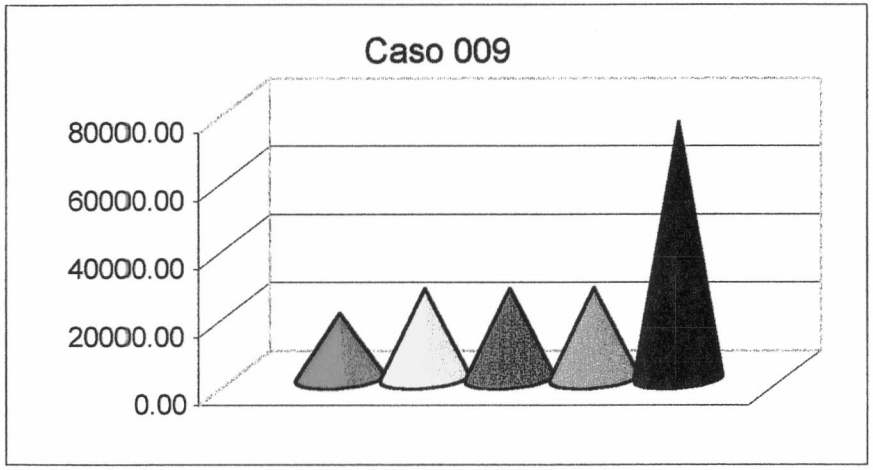

Caso 004
100000.00
80000.00
60000.00
40000.00
20000.00
0.00

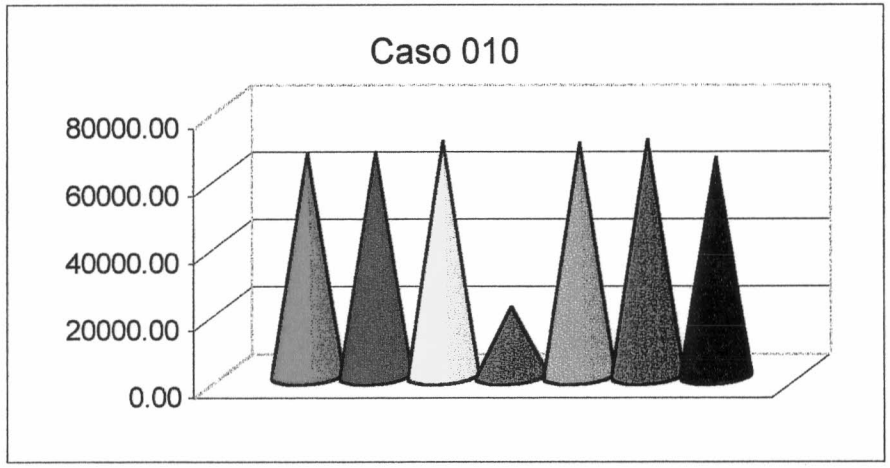

Caso 005
25000.00
24500.00
24000.00
23500.00
23000.00
22500.00
22000.00
21500.00
21000.00
20500.00
20000.00
19500.00

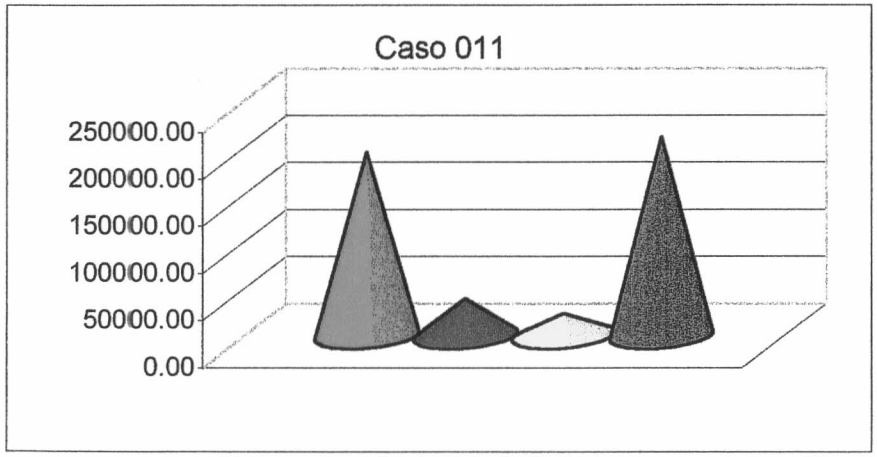

Caso 006
25000.00
20000.00
15000.00
10000.00
5000.00
0.00

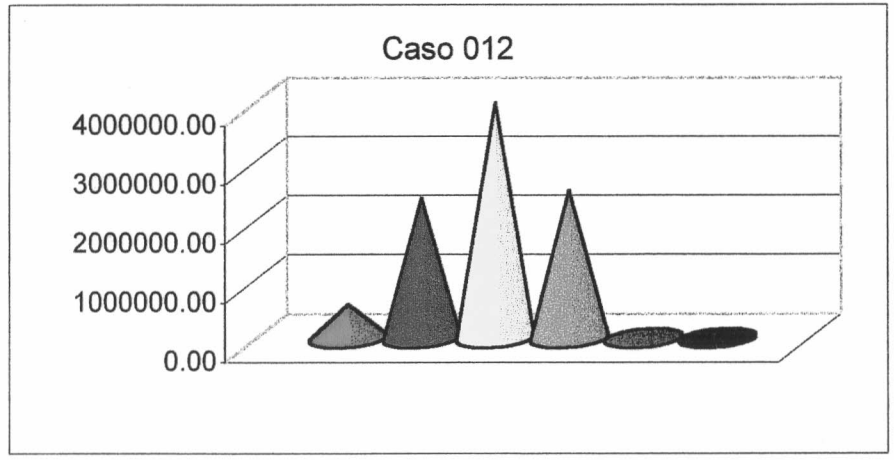

Caso 007
250000.00
200000.00
150000.00
100000.00
50000.00
0.00

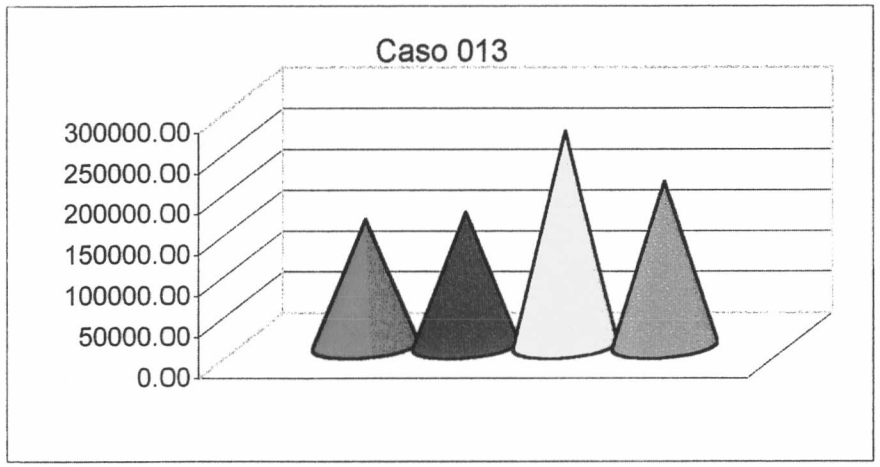

Caso 008
400000.00
300000.00
200000.00
100000.00
0.00

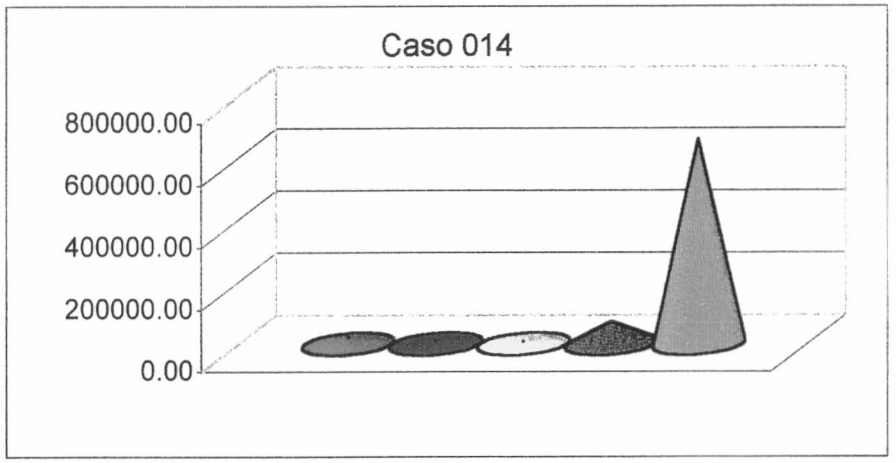

Caso 009
80000.00
60000.00
40000.00
20000.00
0.00

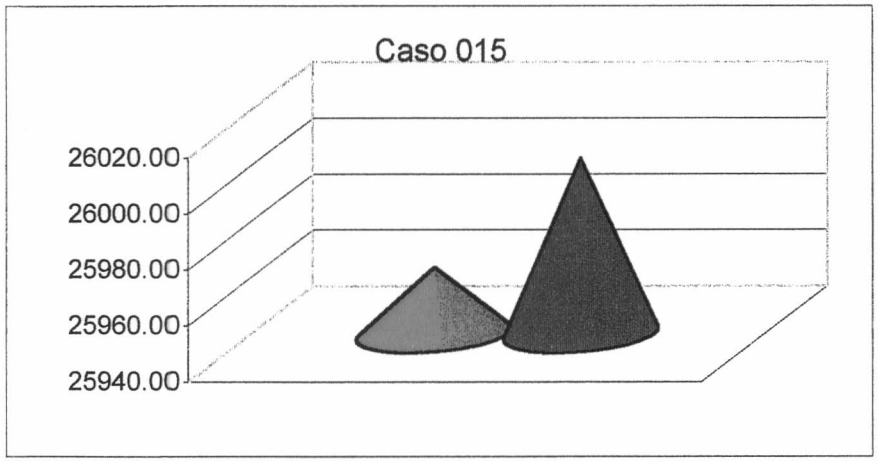

Caso 010
80000.00
60000.00
40000.00
20000.00
0.00

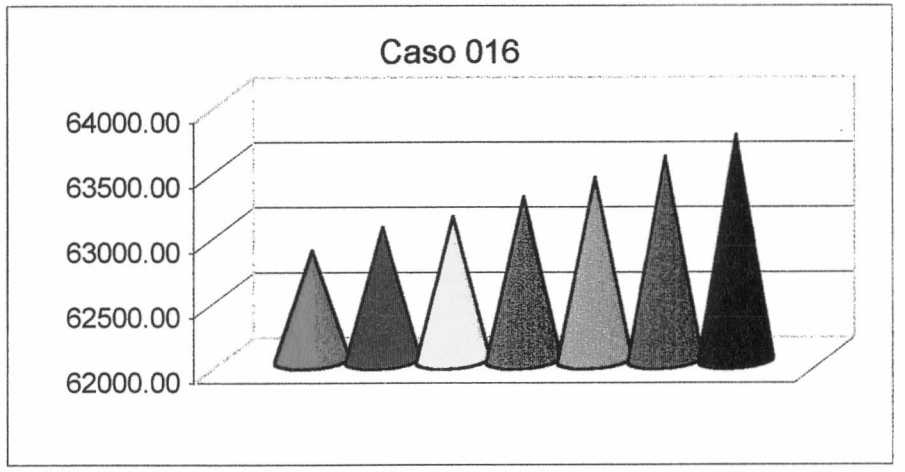

Caso 011
250000.00
200000.00
150000.00
100000.00
50000.00
0.00

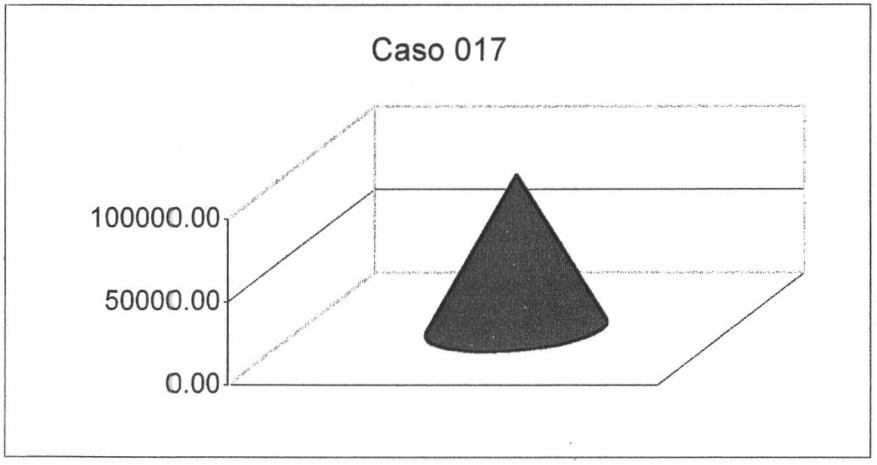

Caso 012
4000000.00
3000000.00
2000000.00
1000000.00
0.00

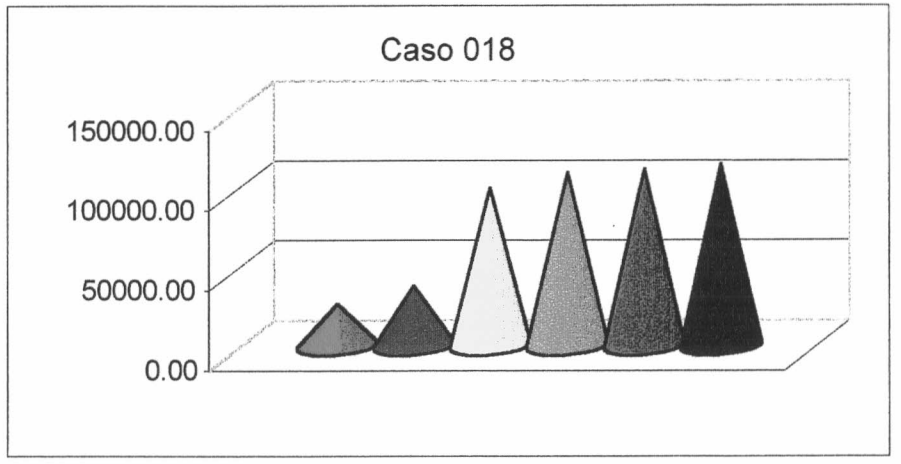

Caso 013
300000.00
250000.00
200000.00
150000.00
100000.00
50000.00
0.00

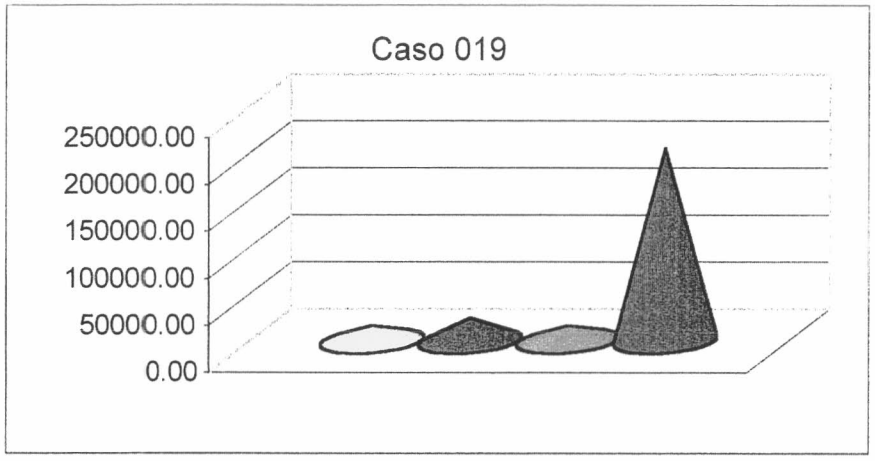

Caso 014
800000.00
600000.00
400000.00
200000.00
0.00

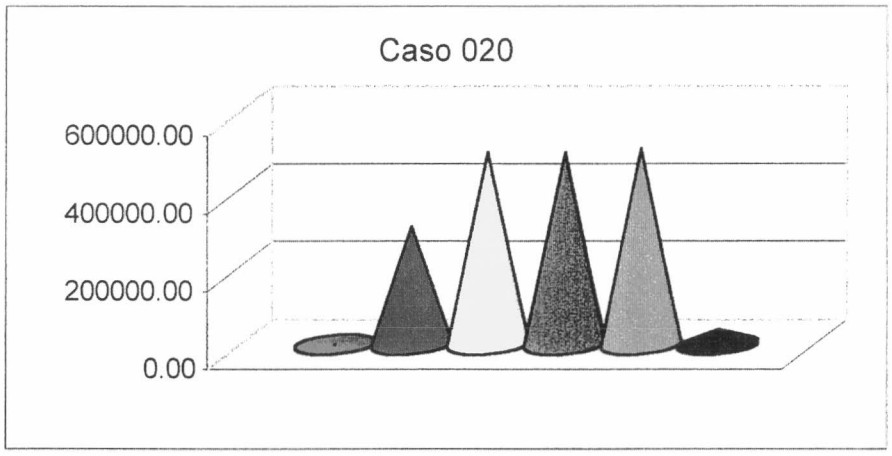

Caso 015
26020.00
26000.00
25980.00
25960.00
25940.00

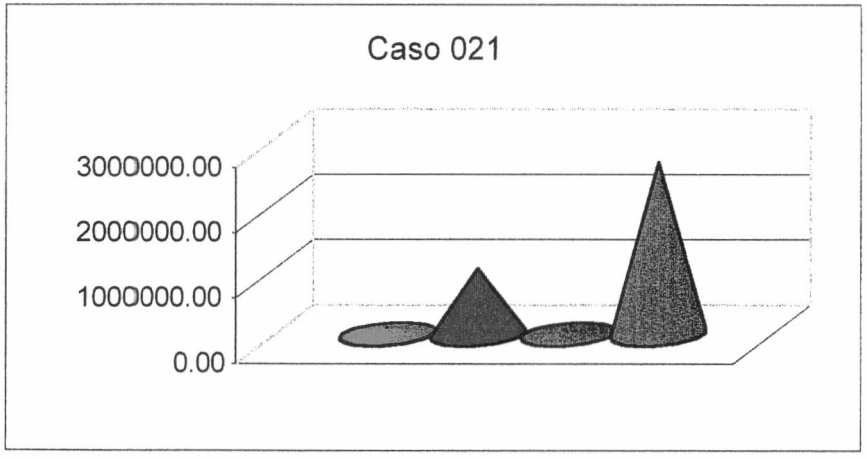

Caso 016
64000.00
63500.00
63000.00
62500.00
62000.00

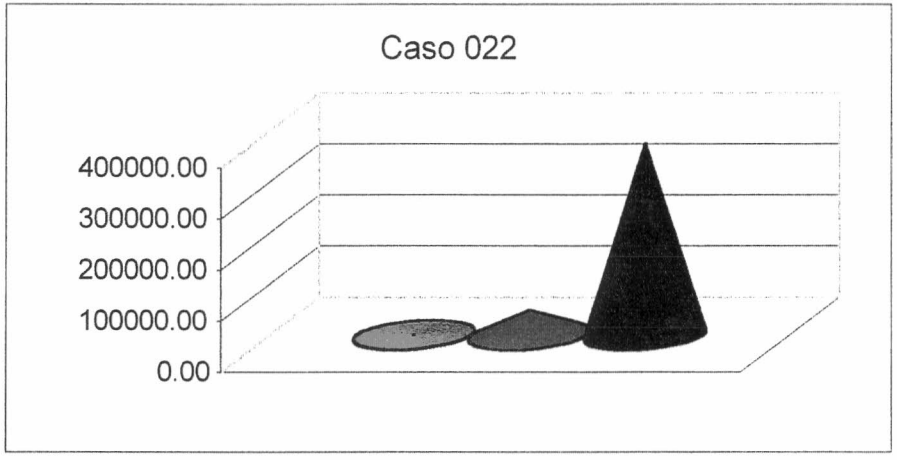

Caso 017
100000.00
50000.00
0.00

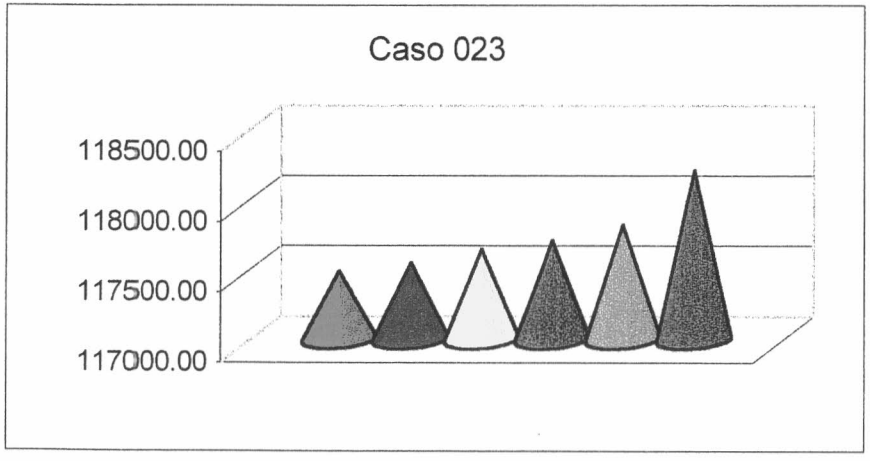

Caso 018
150000.00
100000.00
50000.00
0.00

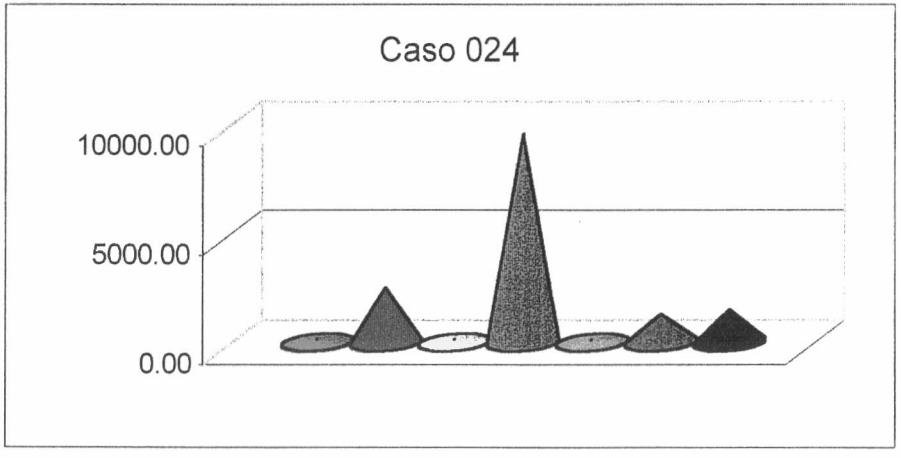

Caso 019
250000.00
200000.00
150000.00
100000.00
50000.00
0.00

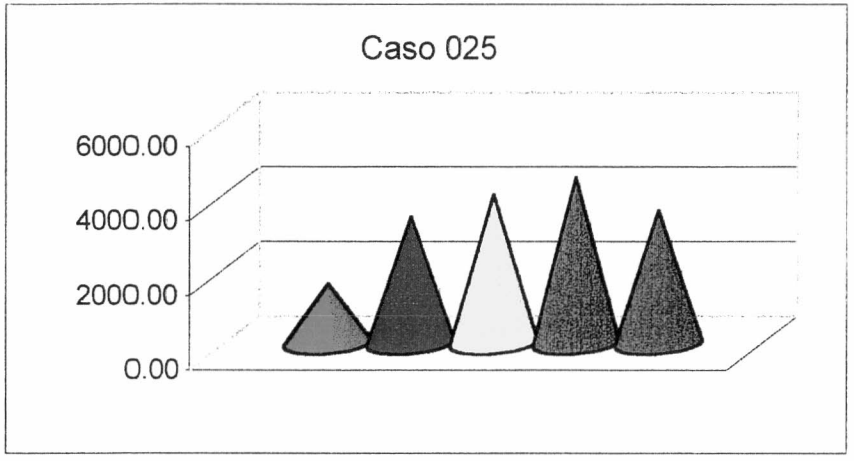

Caso 020
600000.00
400000.00
200000.00
0.00

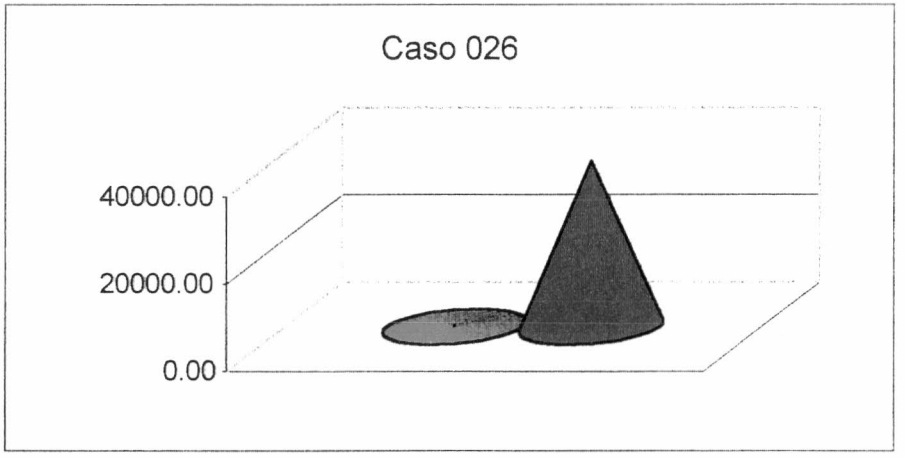

Caso 021
3000000.00
2000000.00
1000000.00
0.00


Caso 022
400000.00
300000.00
200000.00
100000.00
0.00


Caso 023
118500.00
118000.00
117500.00
117000.00


Caso 024
10000.00
5000.00
0.00


Caso 025
6000.00
4000.00
2000.00
0.00

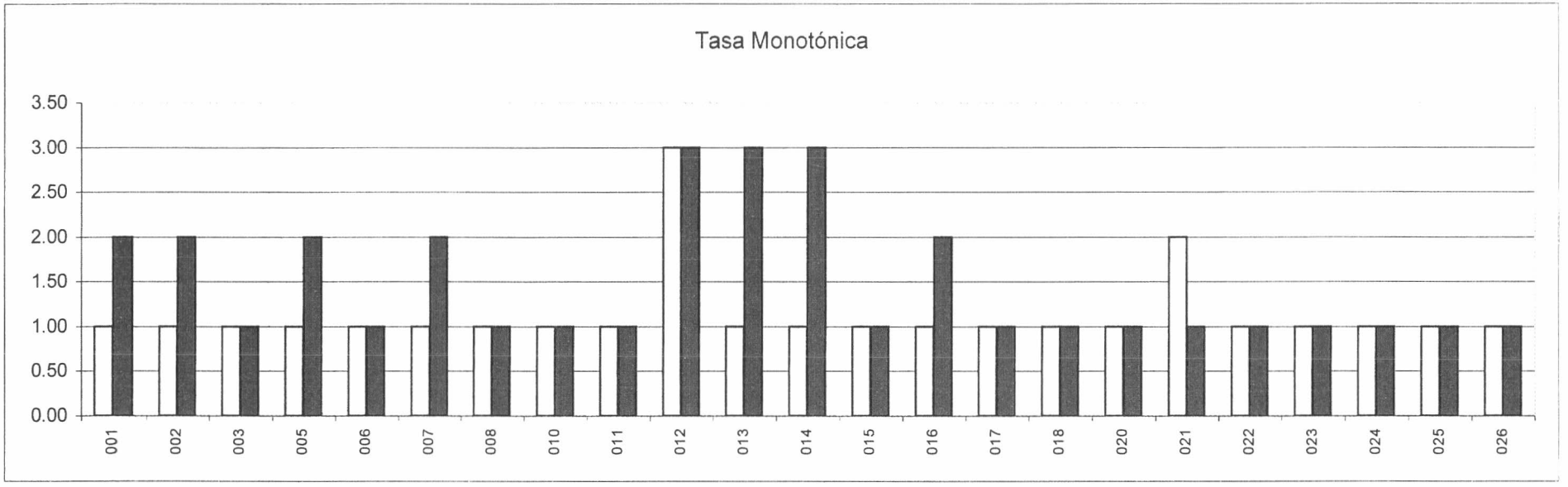

Caso 026
40000.00
20000.00
0.00

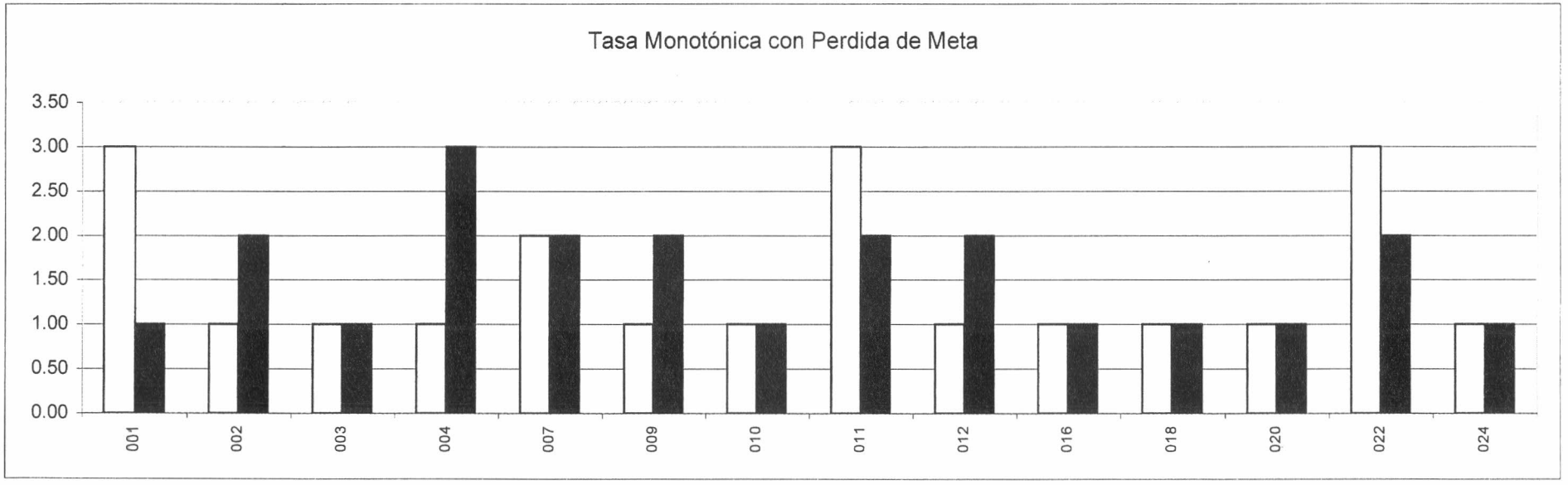
Tiempo Ocioso – Metas Periódicas Perdidas

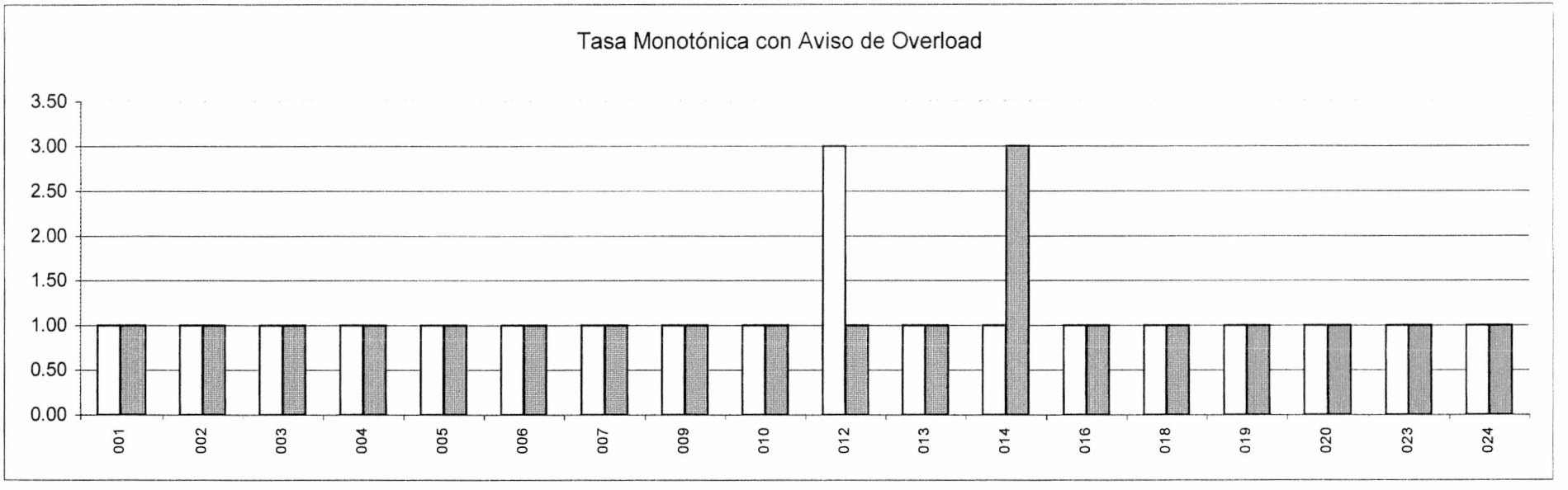
Los siguientes gráficos muestran una comparación del tiempo ocioso del procesador con la cantidad de metas perdidas de los algoritmos en cada prueba realizada. Para que esta comparación fuera mensurable se dividió el universo de los valores de tiempo ocioso y de metas perdidas en tres, los menores a 1/3 del universo los que estaban entres 1/3 y 2/3 y los que superaban los 2/3 asignándoles los valores 1, 2 y 3 respectivamente.

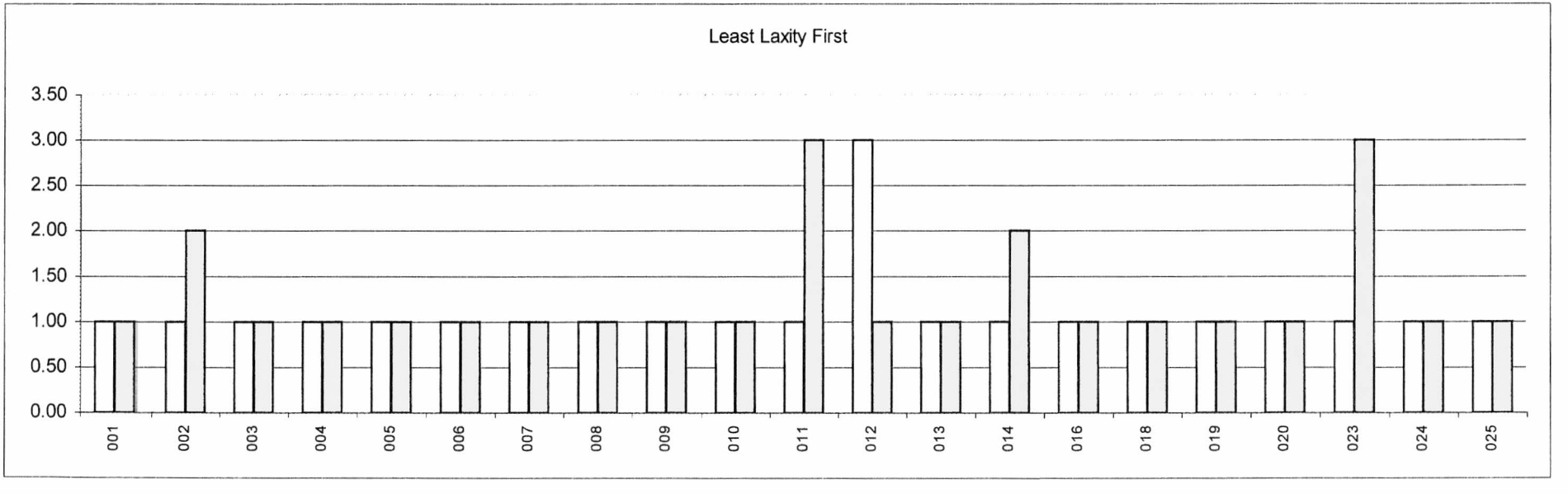
Referencia:
Algoritmos:

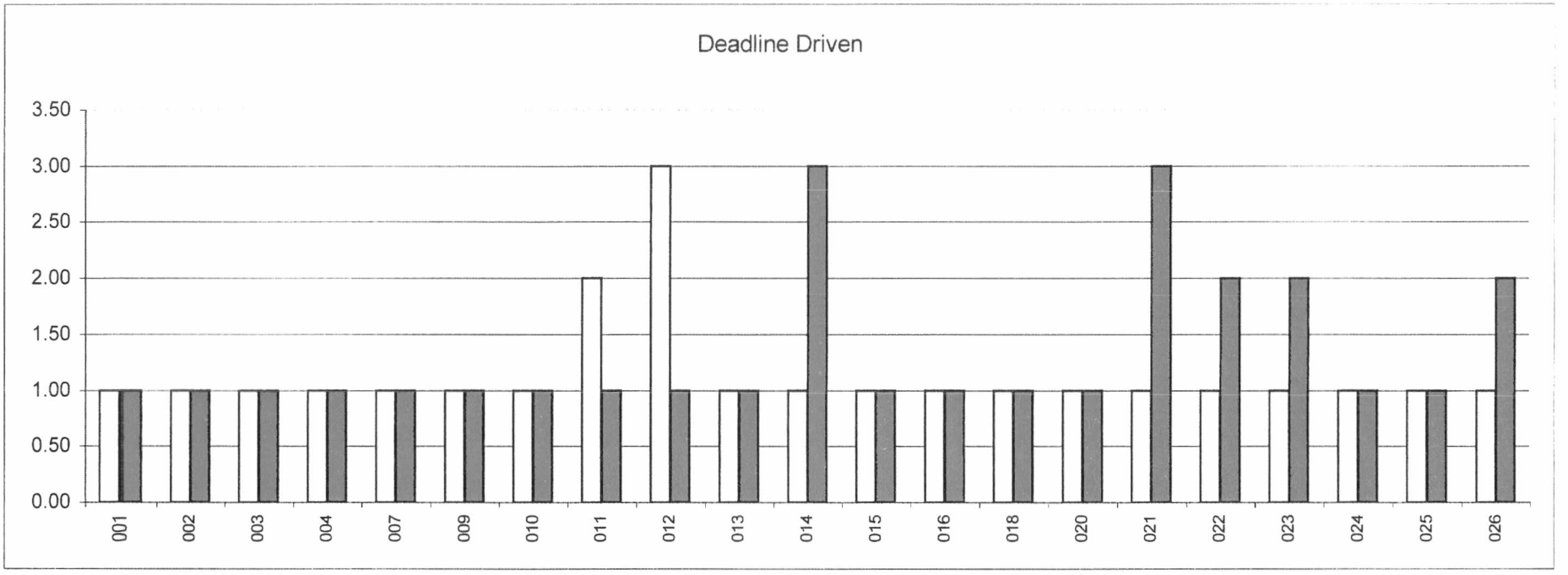
- Deadline Driven
- Tasa Monotónica
- Least Laxity First
- Deadline Driven con Aviso de Overload
- Tasa Monotónica con Aviso de Overload
- Deadline Driven con Aviso de Perdida de Meta
- Tasa Monotónica con Aviso de Perdida de Meta
- Tiempo Ocioso del Procesador

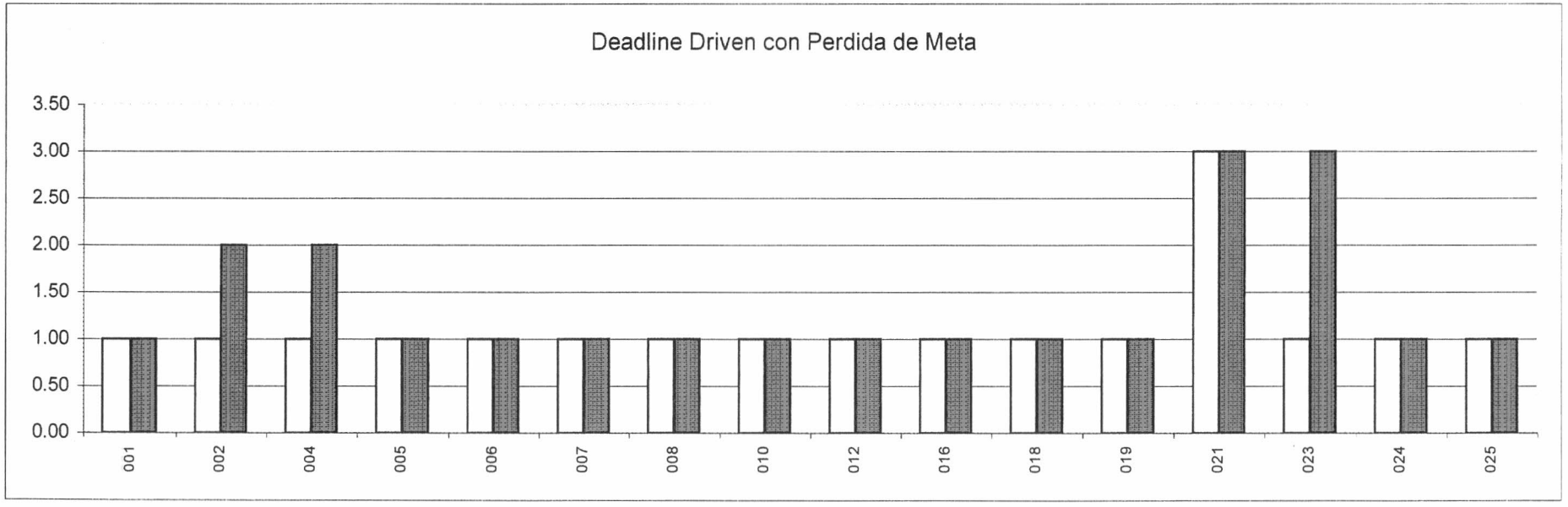

Tasa Monotónica
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
001
002
003
005
006
007
008
010
011
012
013
014
015
016
017
018
020
021
022
023
024
025
026

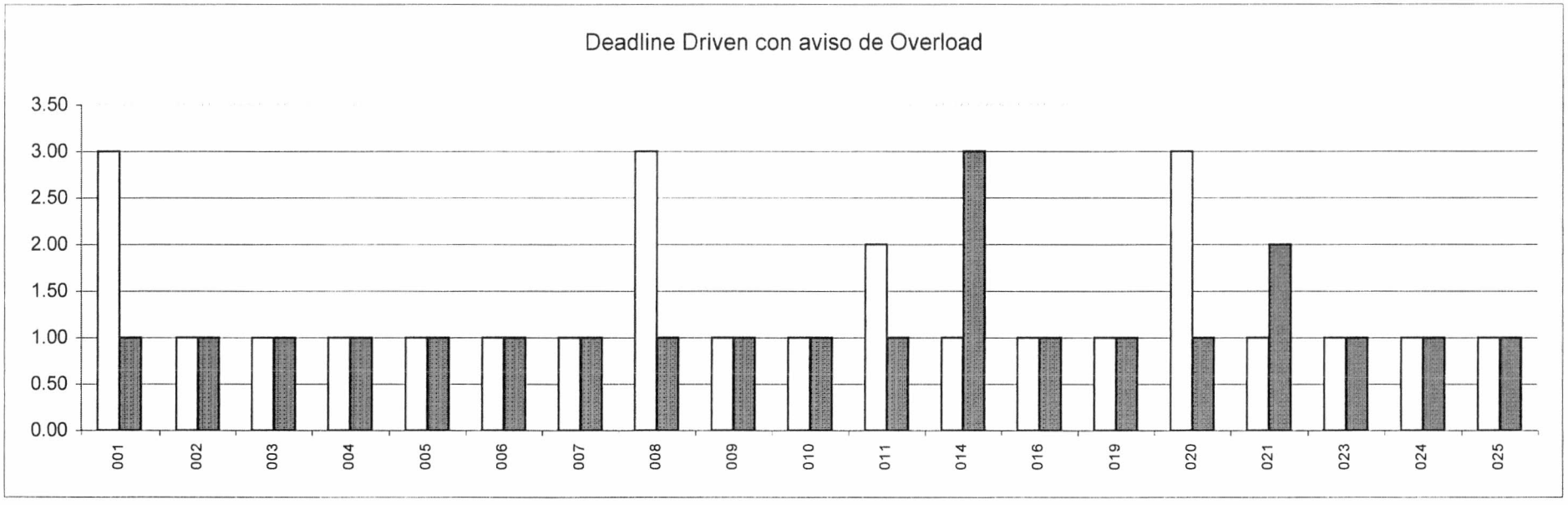

Tasa Monotónica con Perdida de Meta
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
001
002
003
004
007
009
010
011
012
016
018
020
022
024


Tasa Monotónica con Aviso de Overload
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
001
002
003
004
005
006
007
009
010
012
013
014
016
018
019
020
023
024


Least Laxity First
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
001
002
003
004
005
006
007
008
009
010
011
012
013
014
016
018
019
020
023
024
025


Deadline Driven
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
001
002
003
004
007
009
010
011
012
013
014
015
016
018
020
021
022
023
024
025
026

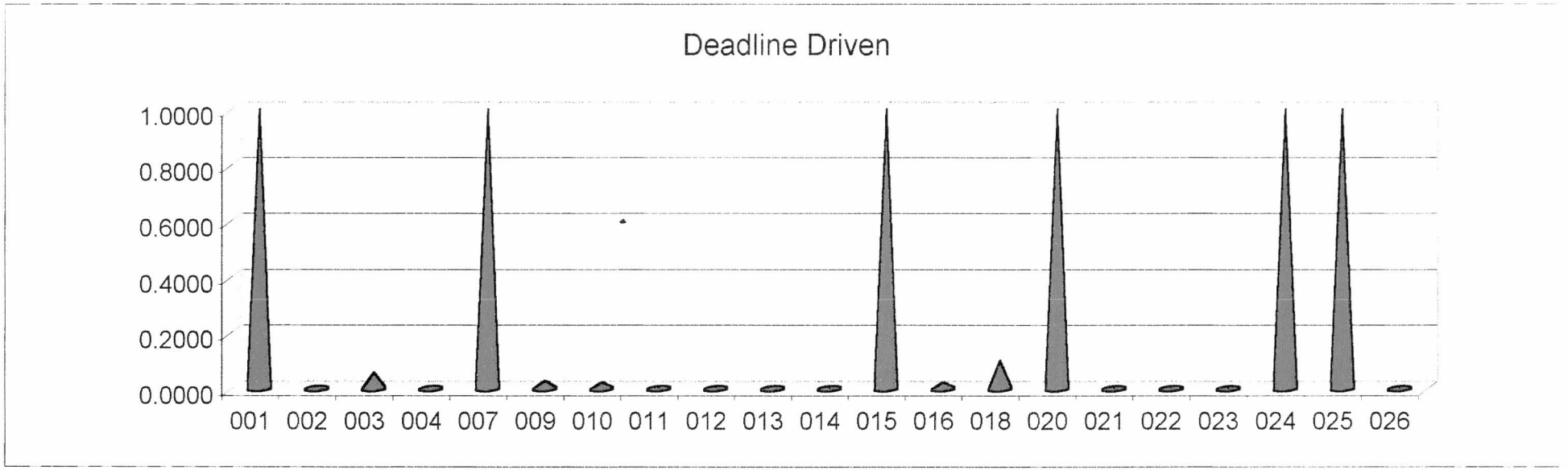

Deadline Driven con Perdida de Meta
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
001
002
004
005
006
007
008
010
012
016
018
019
021
023
024
025

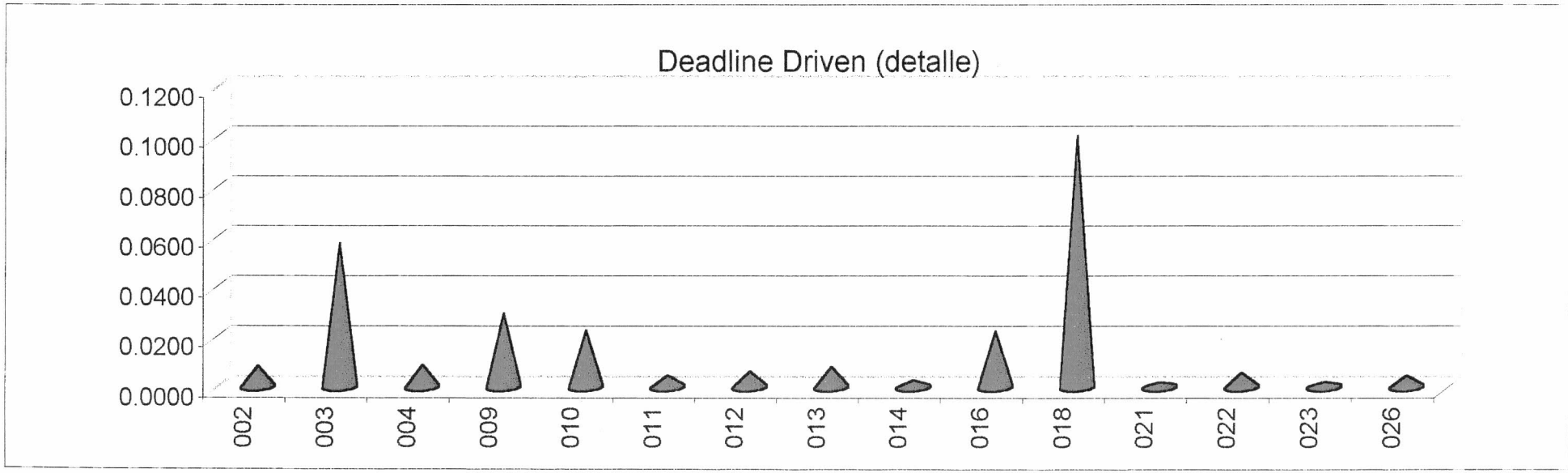

Deadline Driven con aviso de Overload
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
001
002
003
004
005
006
007
008
009
010
011
014
016
019
020
021
023
024
025

Algoritmos – Porcentaje de Garantía

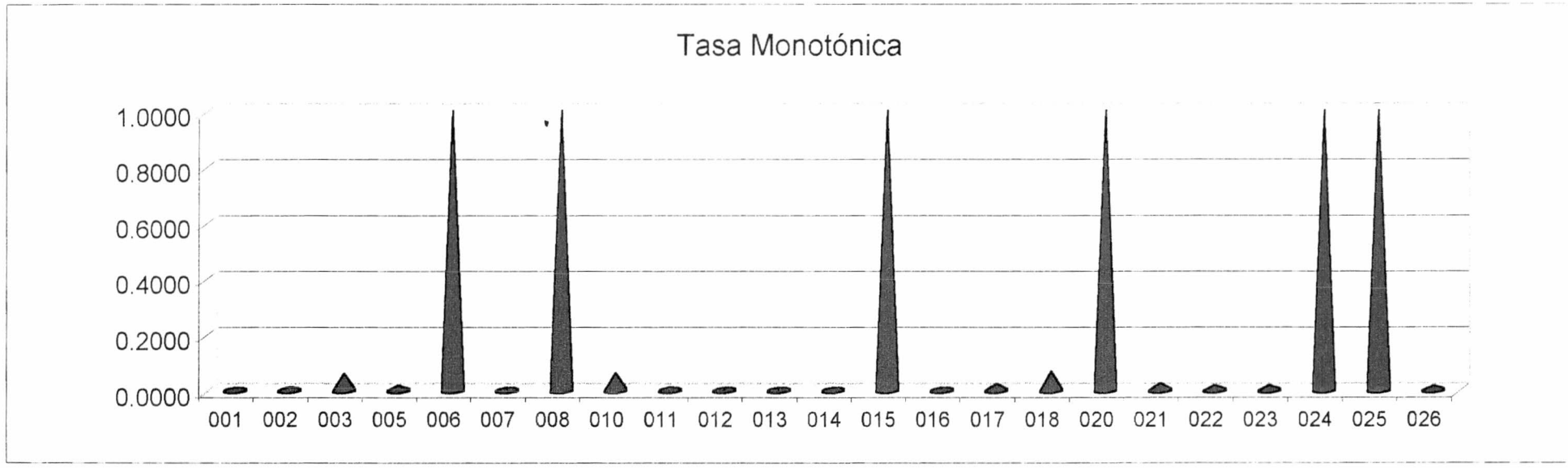
En los siguientes gráficos se muestra una comparación del porcentaje de Garantía de cada algoritmo para los casos de prueba, se muestra una comparación por algoritmo, también se presenta un detalle de aquellos casos que no ofrecieron en 100% de la garantía.

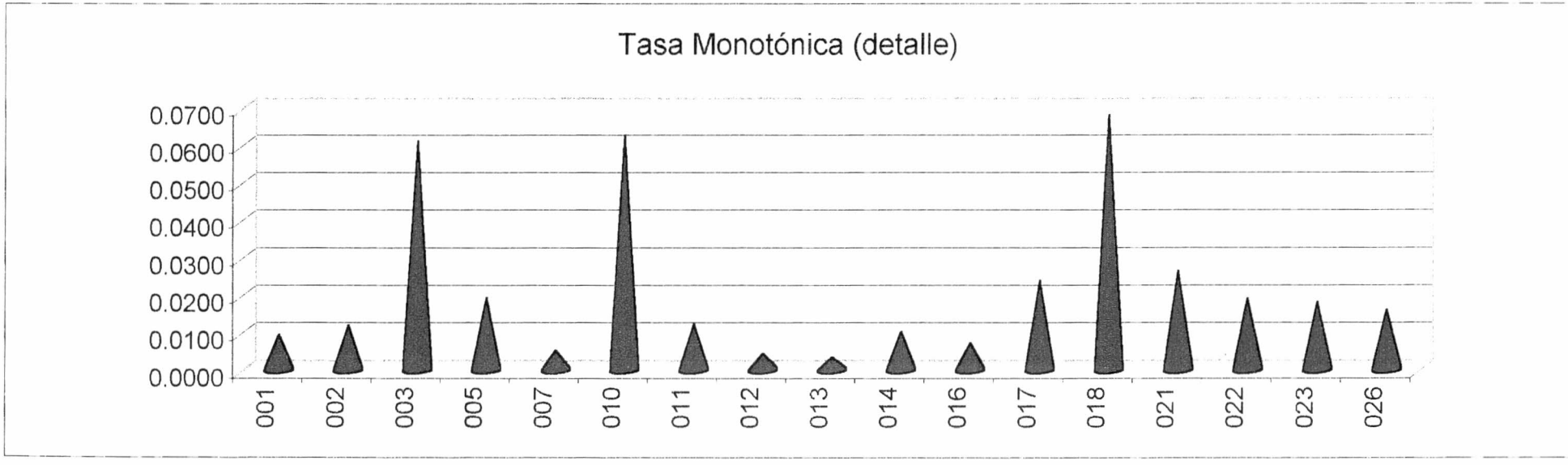
Porcentaje de garantía por algoritmo



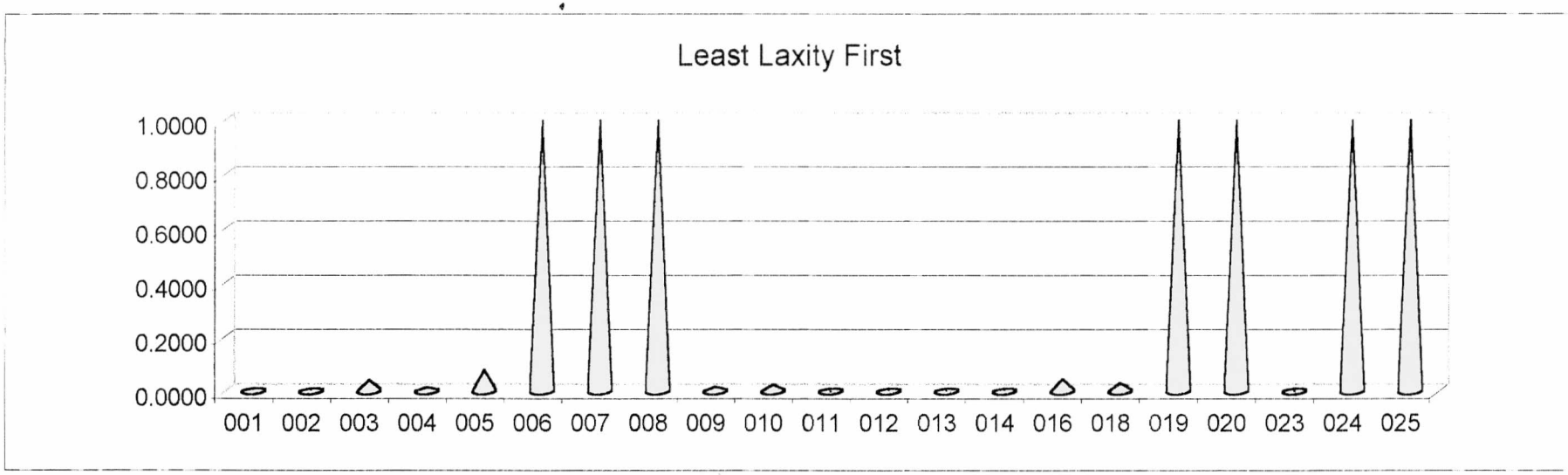


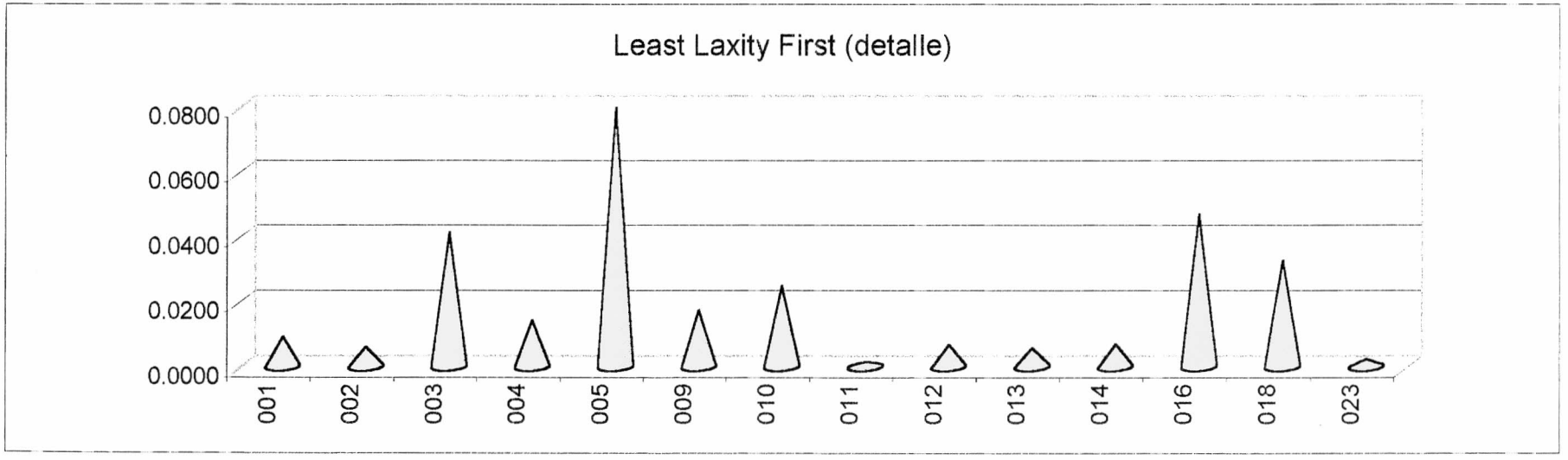
Porcentaje de garantía por algoritmo



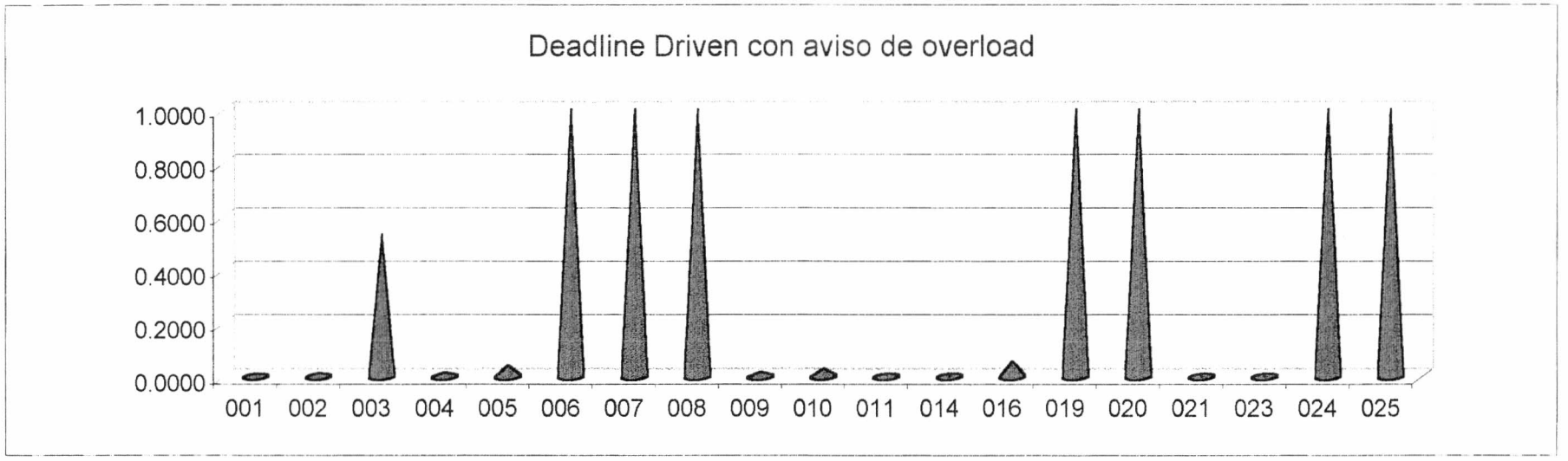


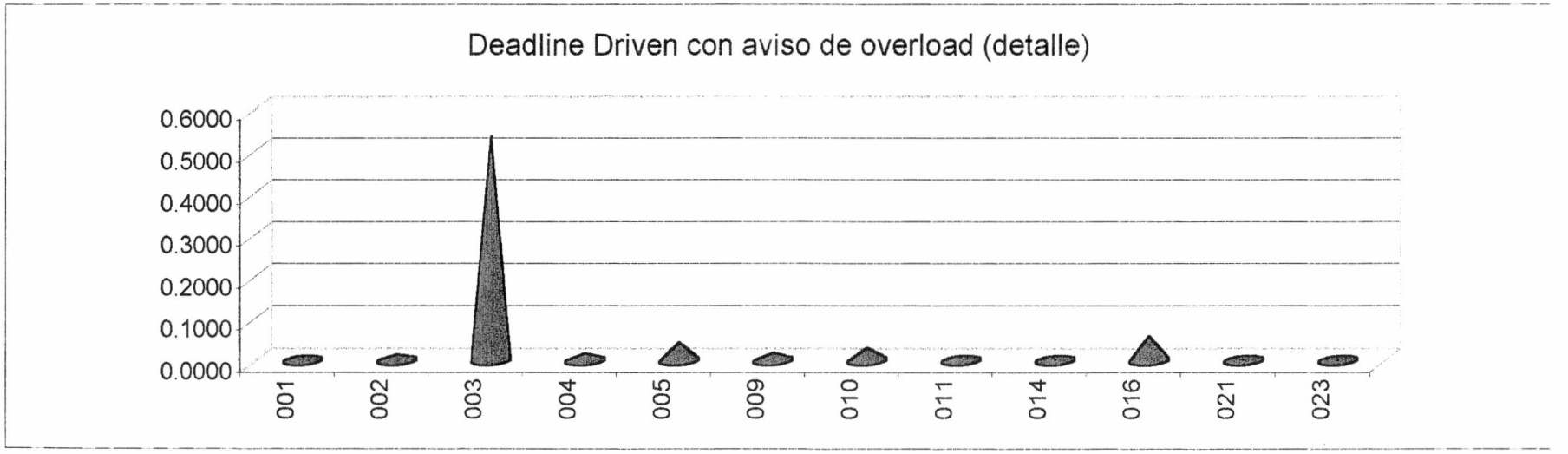
Porcentaje de garantía por algoritmo



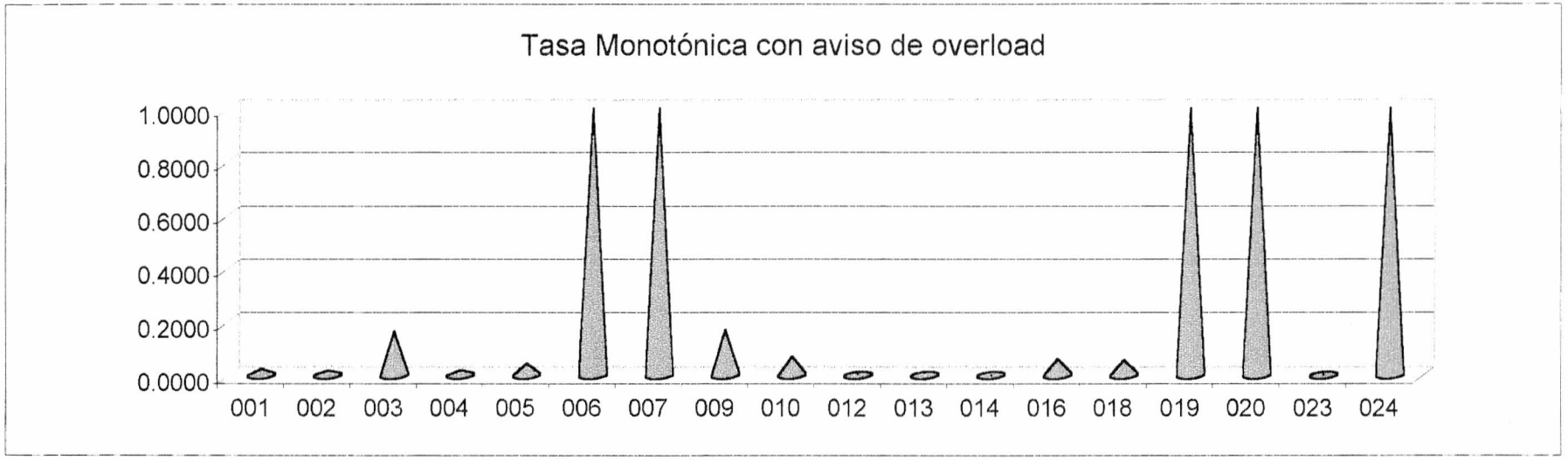


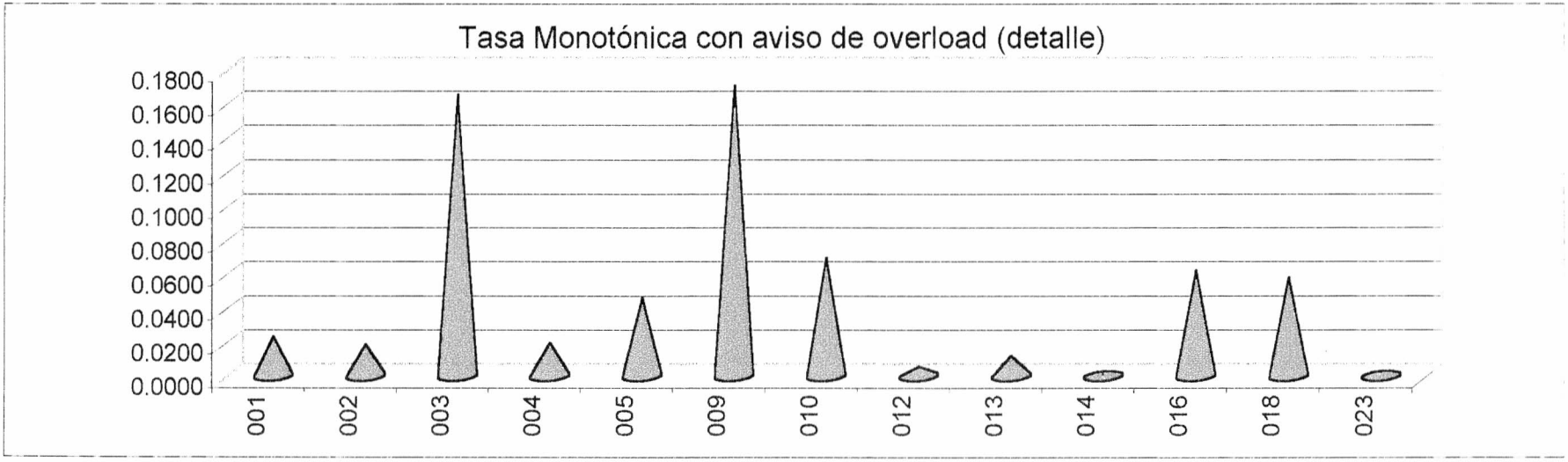
Porcentaje de garantía por algoritmo



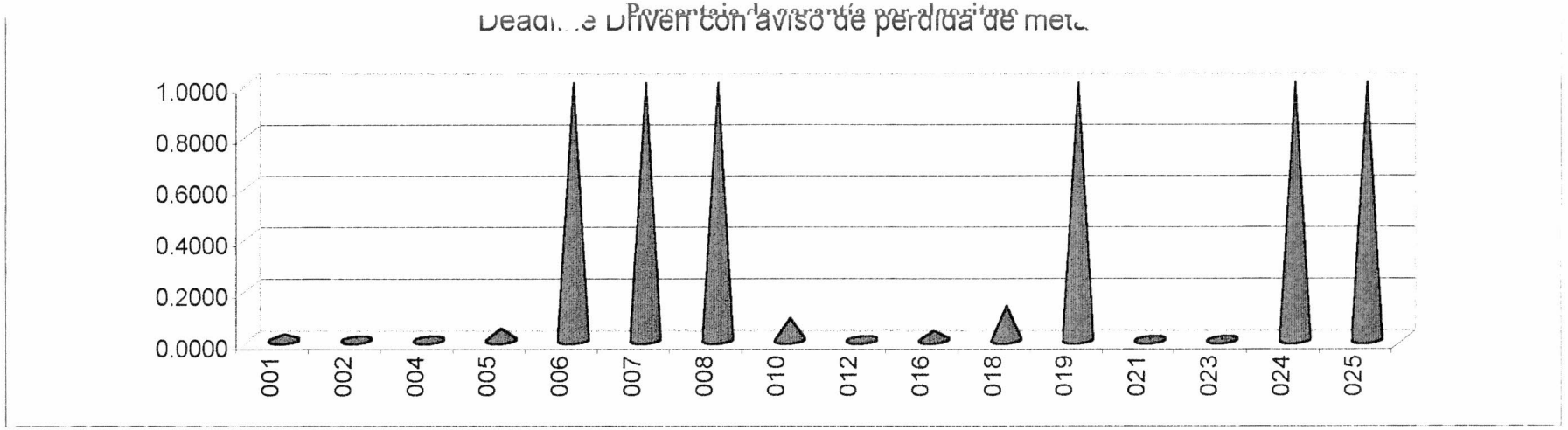


Porcentaje de garantía por algoritmo



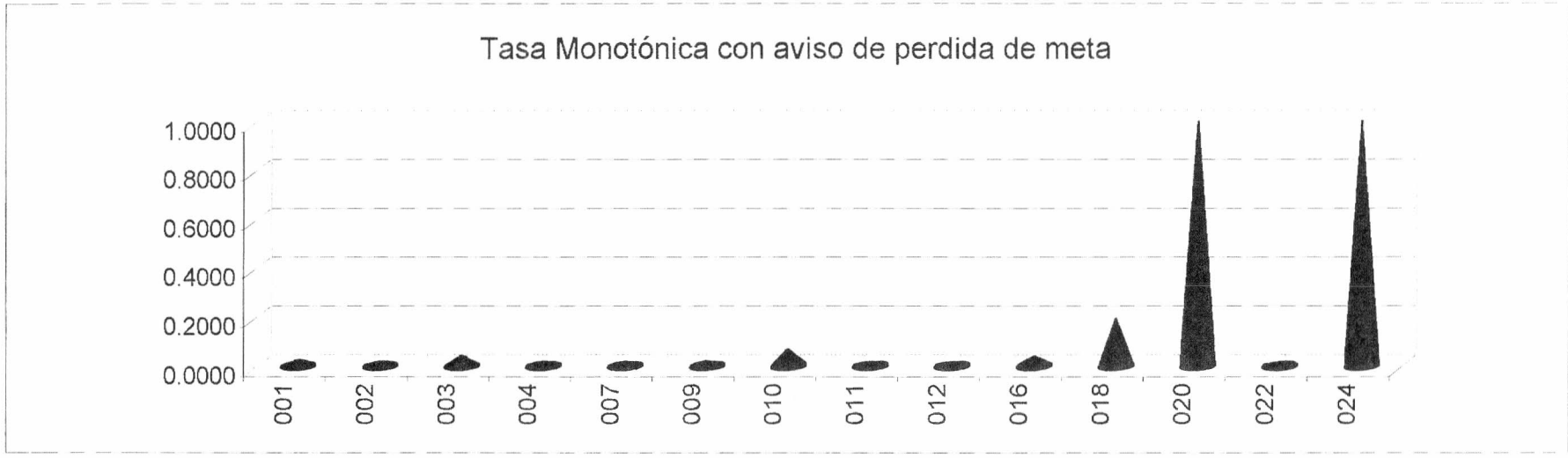


Deadline Driven con aviso de perdida de meta


Deadline Driven con aviso de perdida de meta
1.0000
0.8000
0.6000
0.4000
0.2000
0.0000
001
002
004
005
006
007
008
010
012
016
018
019
021
023
024
025


Deadline Driven con aviso de perdida de meta (detalle)
0.1400
0.1200
0.1000
0.0800
0.0600
0.0400
0.0200
0.0000
001
002
004
005
010
012
016
018
023

Porcentaje de garantía por algoritmo



Lotes de Prueba – Porcentaje de Garantía

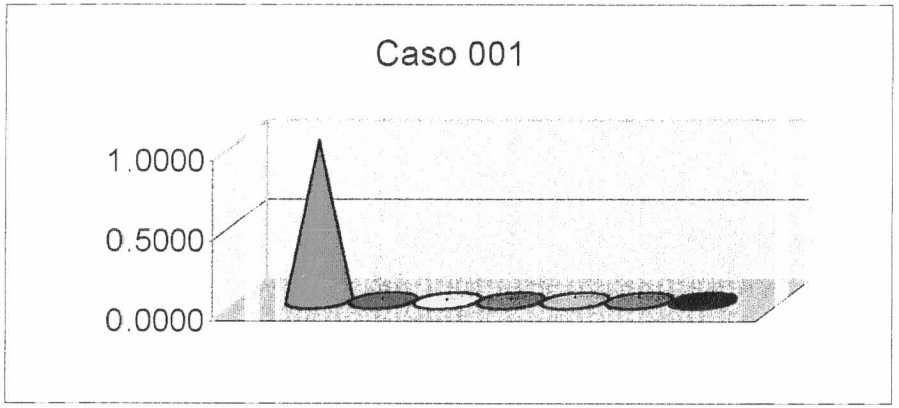
Los siguientes gráficos muestran el porcentaje de garantía de cada caso de prueba con los algoritmos que fueron testeados.

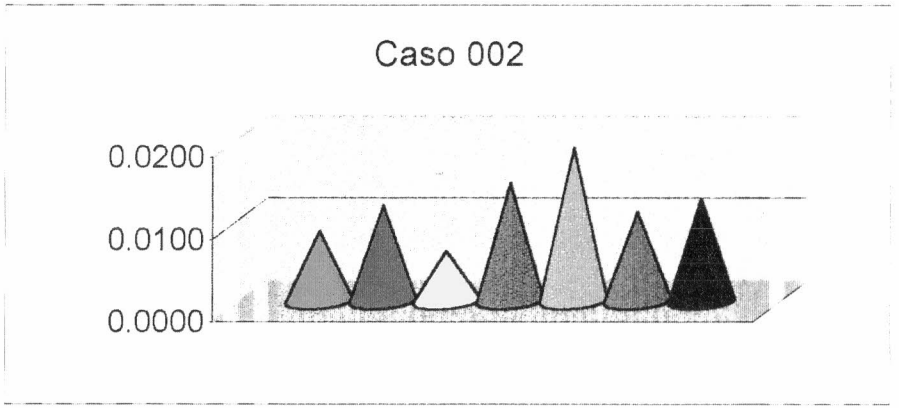
Referencia:

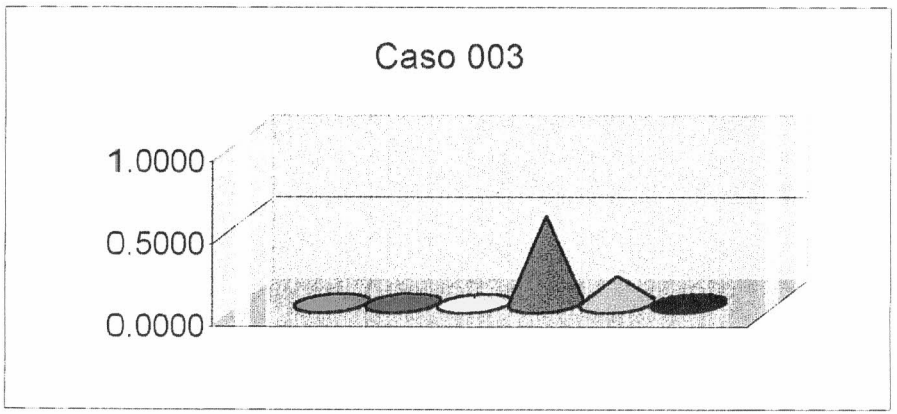
Algoritmos:

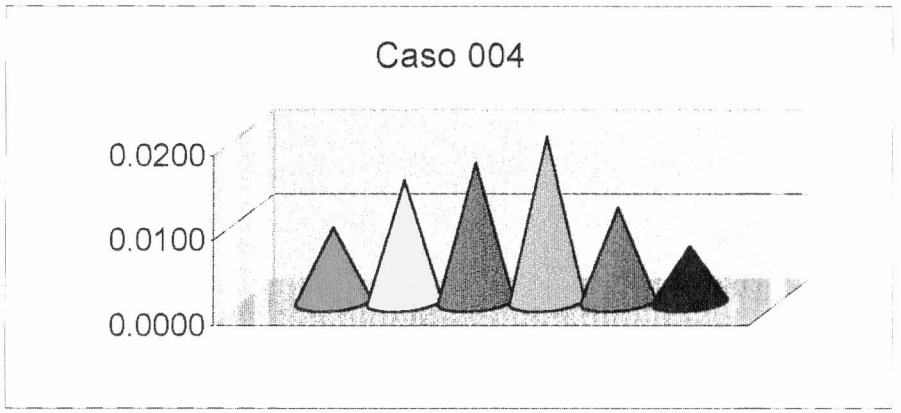
- Deadline Driven
- Tasa Monotónica
- Least Laxity First
- Deadline Driven con Aviso de Overload
- Tasa Monotónica con Aviso de Overload
- Deadline Driven con Aviso de Perdida de Meta
- Tasa Monotónica con Aviso de Perdida de Meta

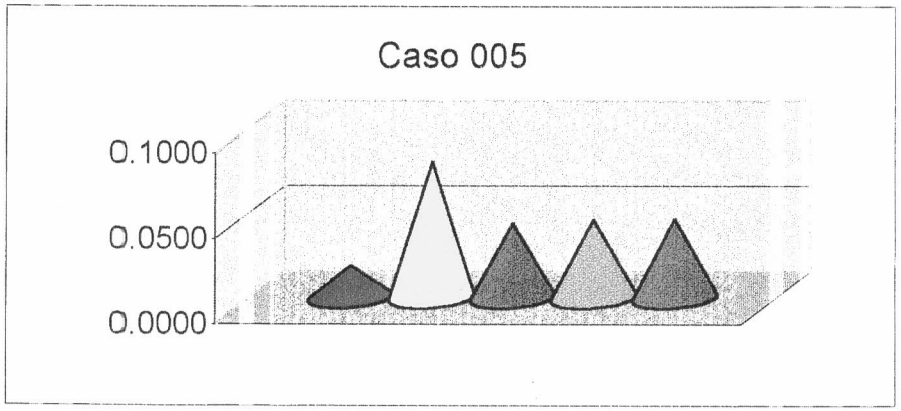
Porcentaje de Garantía

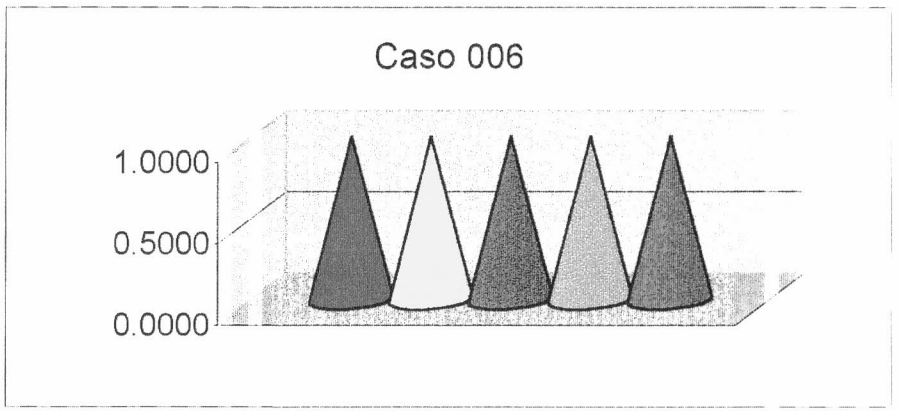




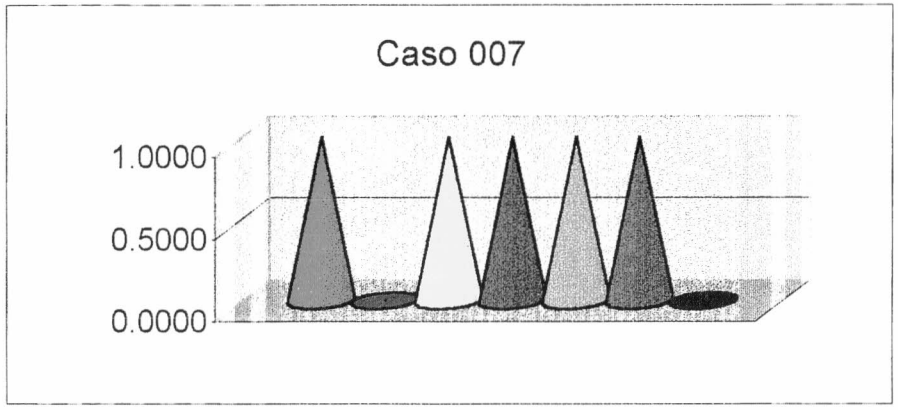


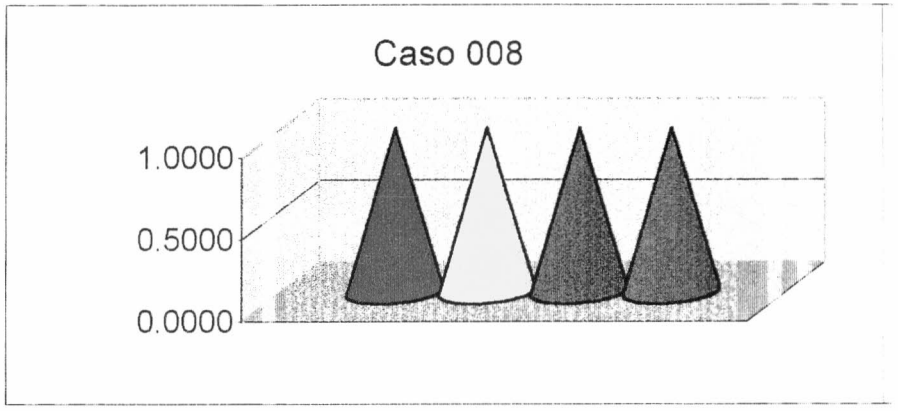


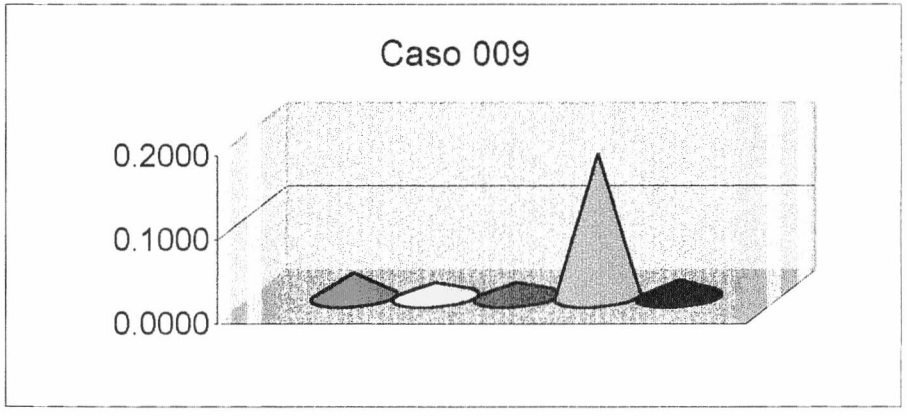


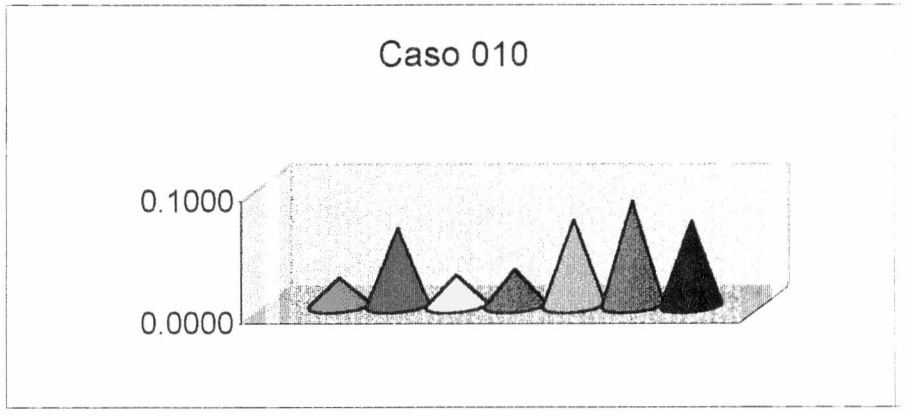


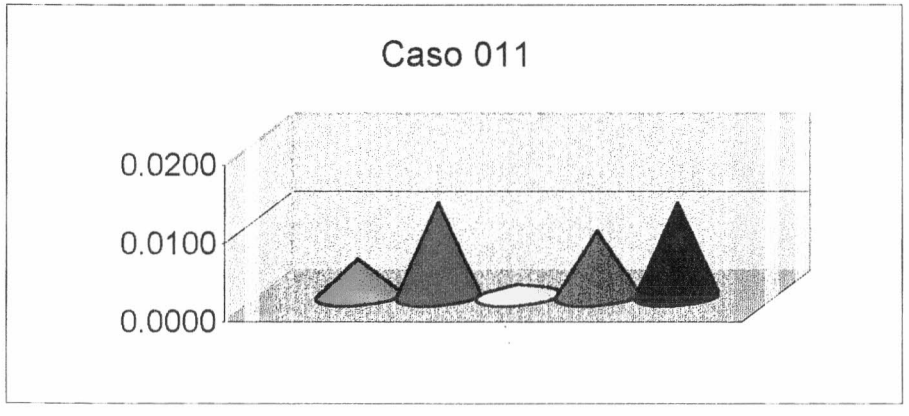
Porcentaje de Garantía

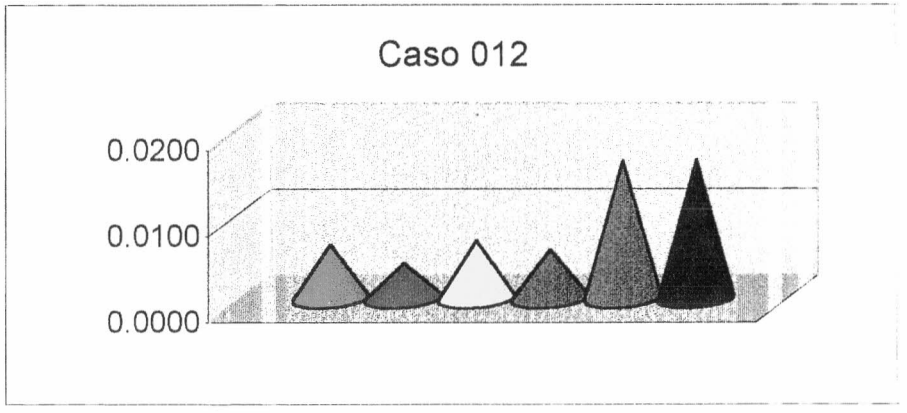




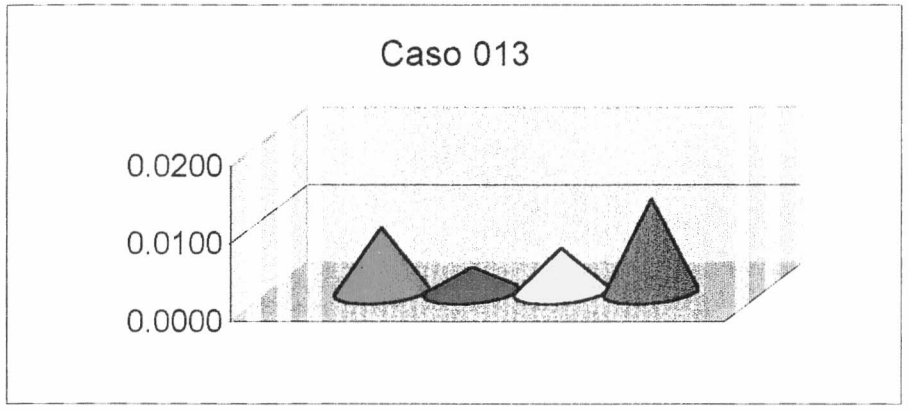


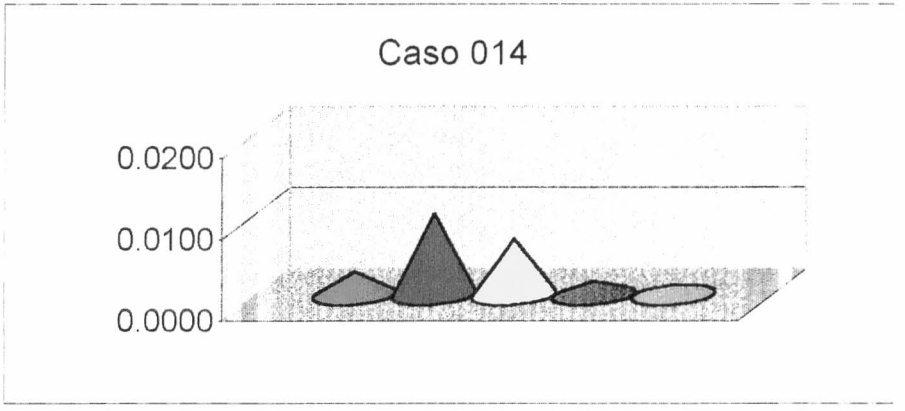


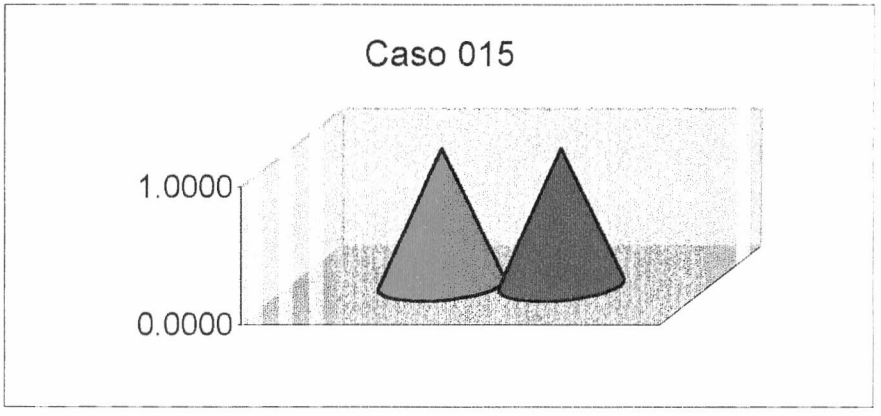


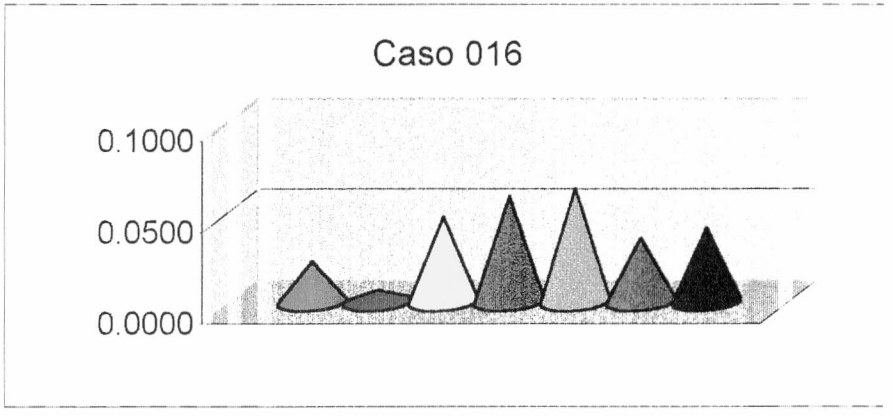


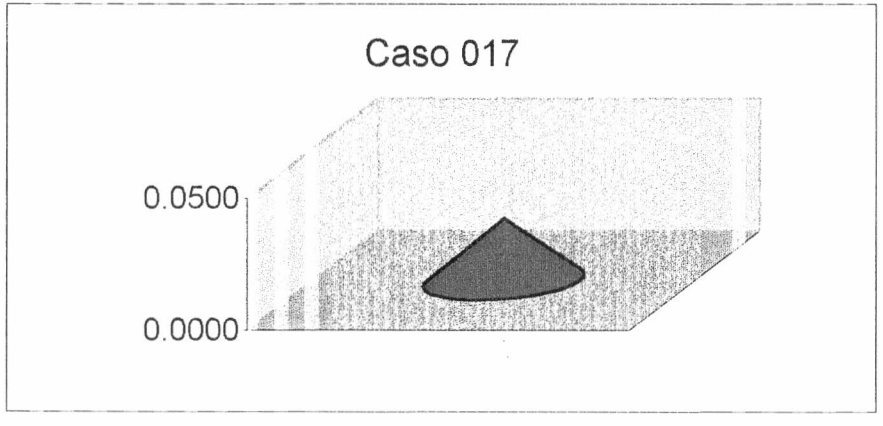
Porcentaje de Garantía

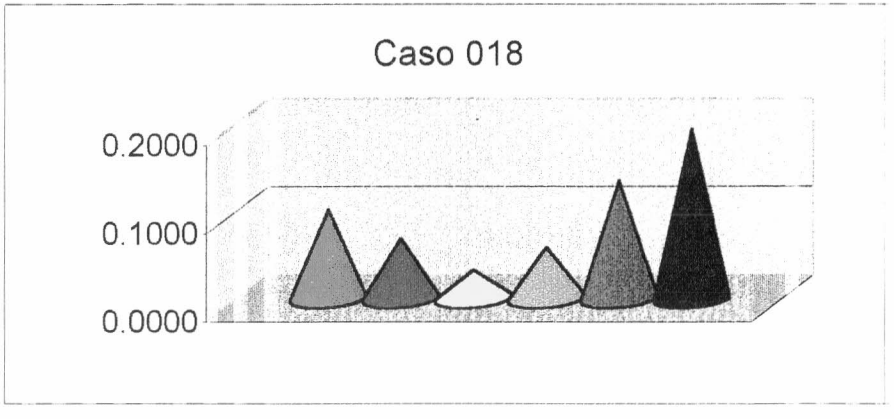




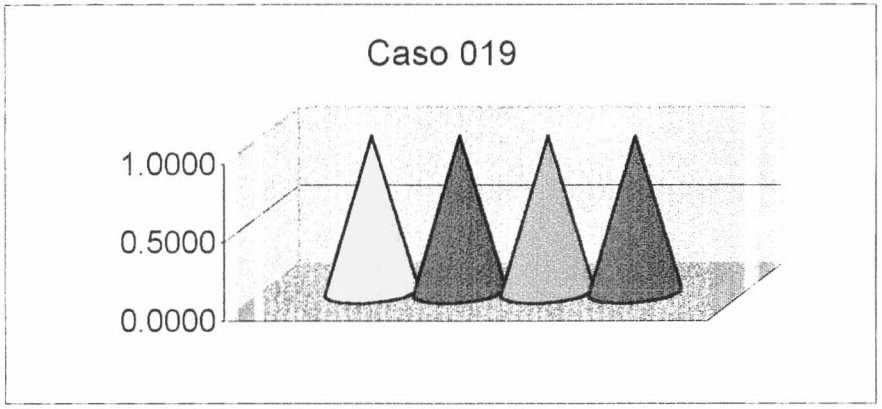


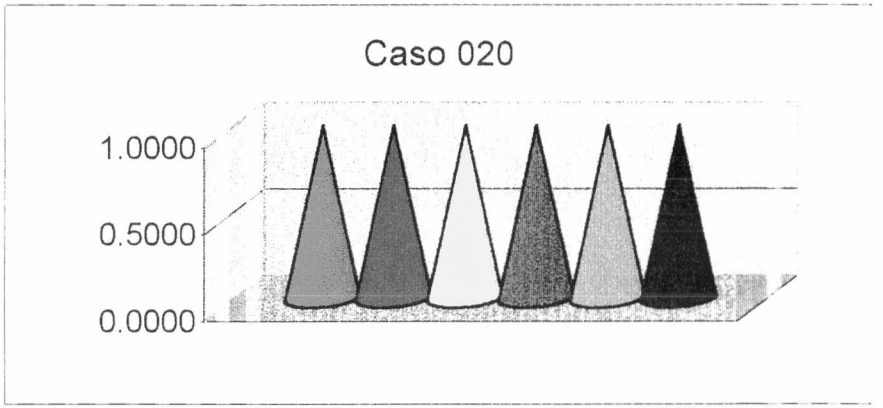


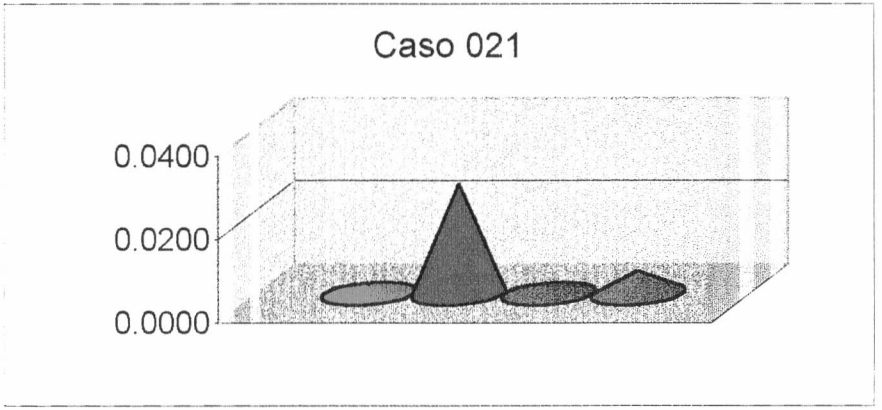


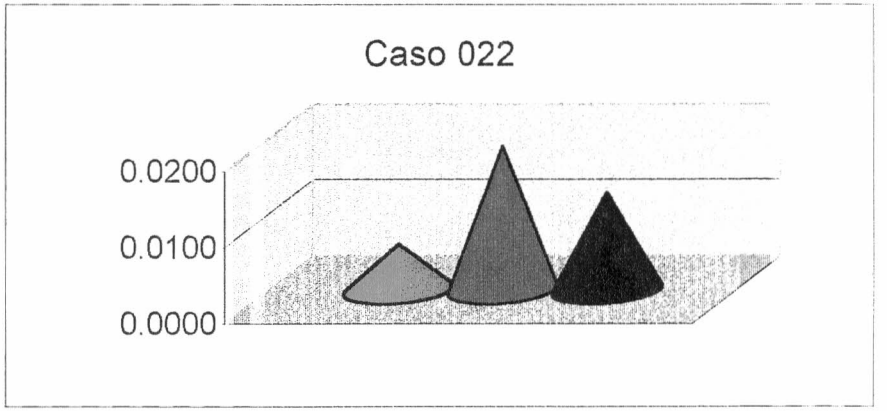


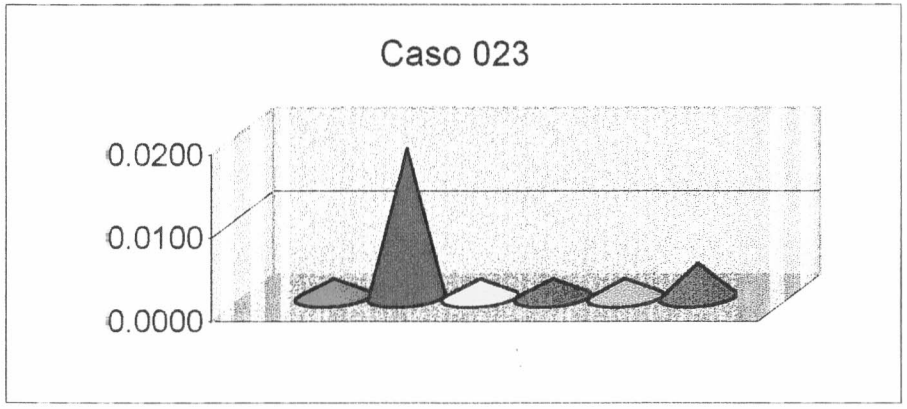
Porcentaje de Garantía

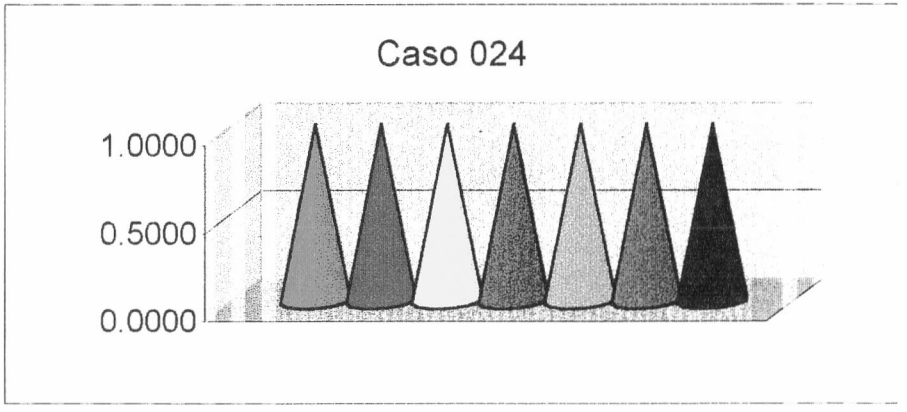




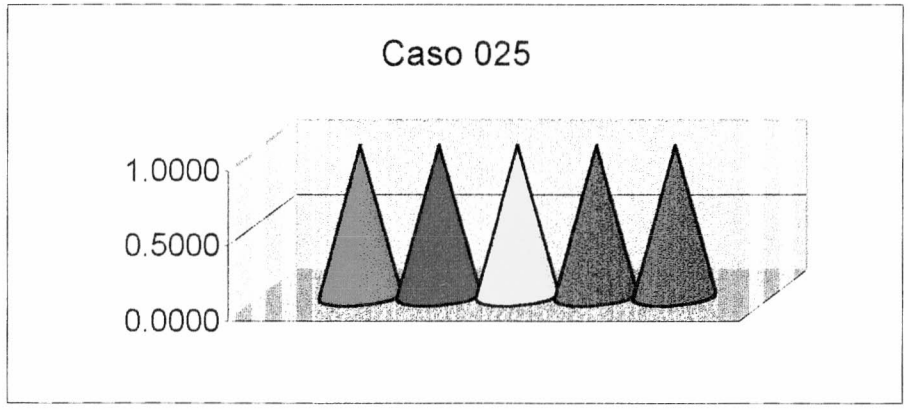


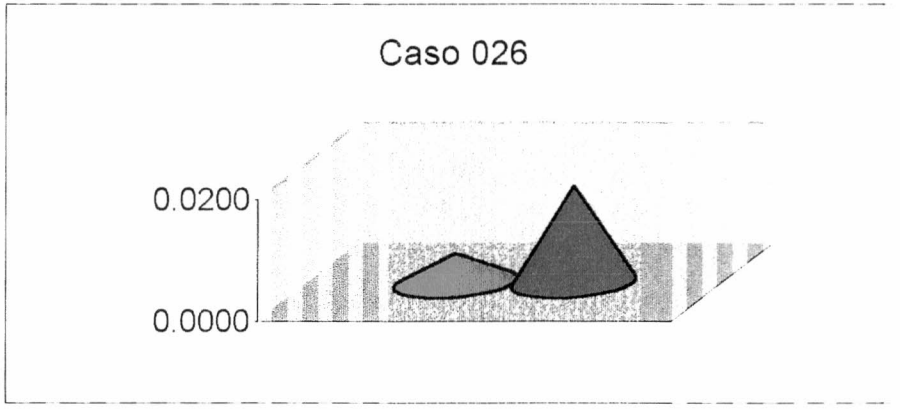






Porcentaje de Garantía



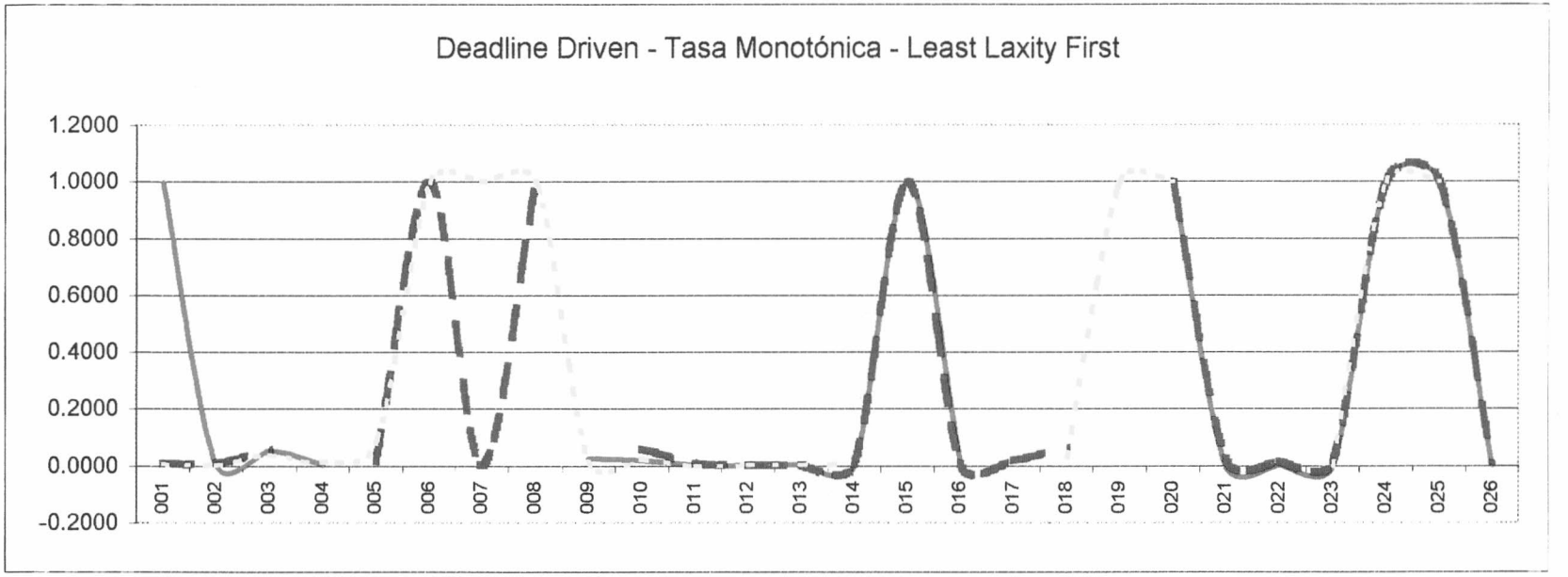


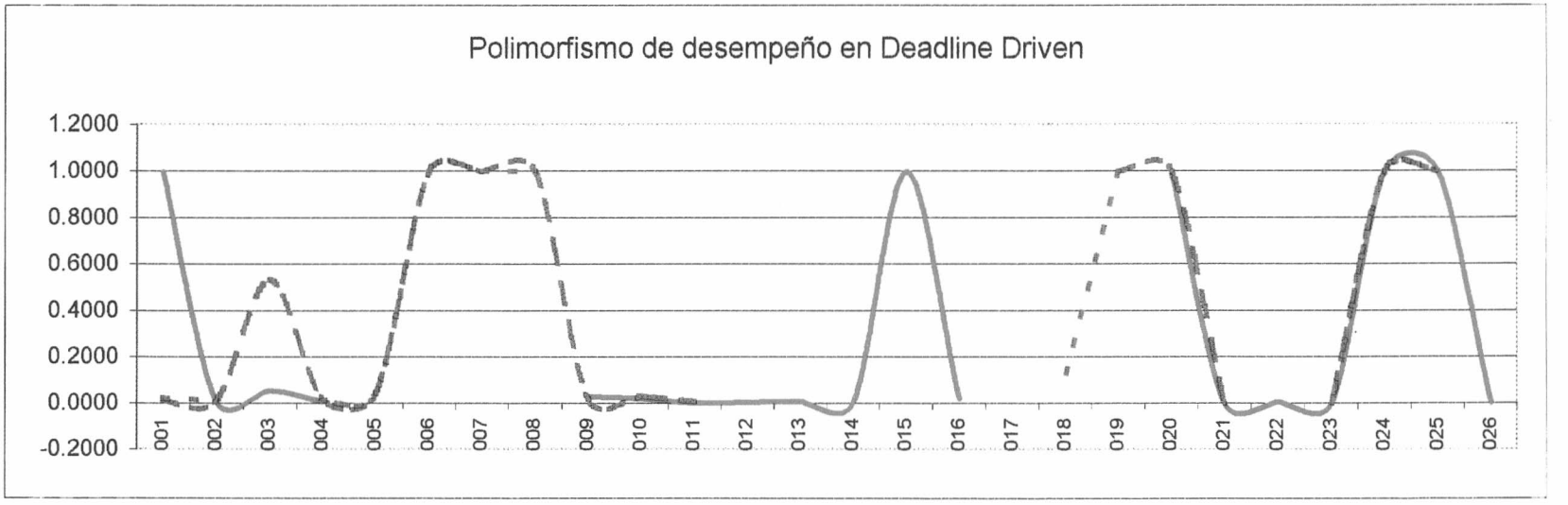
Comparación de la estabilidad de los algoritmos testeados

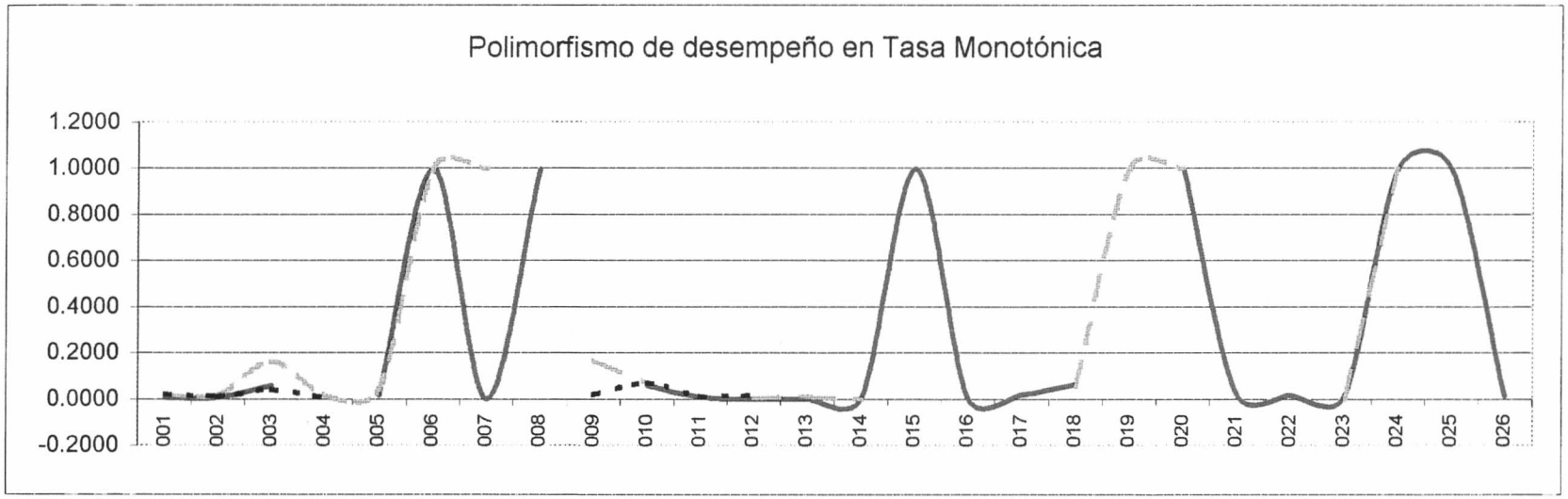
Los siguientes gráficos muestran una comparación del porcentaje de garantía que ofreció cada algoritmo con los casos testeados. El primer gráfico compara los tres algoritmos principales (Tasa Monotónica, Deadline Driven, Least Laxity First). El segundo y tercer gráfico compara la estabilidad de Deadline Driven y Tasa Monotónica respectivamente con respecto a las dos funciones de polimorfismo de desempeño

Referencia:
Algoritmos:

- Deadline Driven
- Tasa Monotónica
- Least Laxity First
- Deadline Driven con Aviso de Overload
- Tasa Monotónica con Aviso de Overload
- Deadline Driven con Aviso de Perdida de Meta
- Tasa Monotónica con Aviso de Perdida de Meta


Deadline Driven - Tasa Monotónica - Least Laxity First
1.2000
1.0000
0.8000
0.6000
0.4000
0.2000
0.0000
-0.2000
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026


Polimorfismo de desempeño en Deadline Driven
1.2000
1.0000
0.8000
0.6000
0.4000
0.2000
0.0000
-0.2000
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026


Polimorfismo de desempeño en Tasa Monotónica
1.2000
1.0000
0.8000
0.6000
0.4000
0.2000
0.0000
-0.2000
001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026

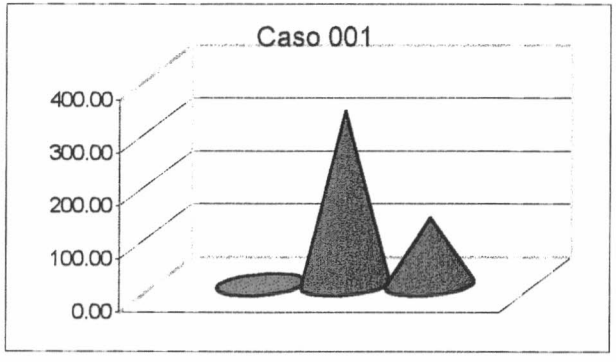
Polimorfismo de Desempeño – Deadline Driven

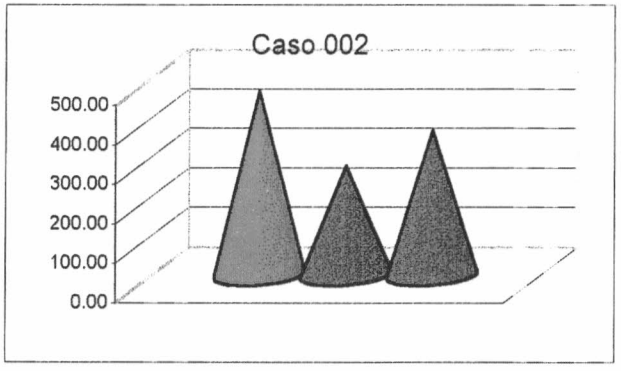
Los siguientes gráficos muestran la relación entre Deadline Driven y las dos funciones de polimorfismo que se testearon para cada caso.

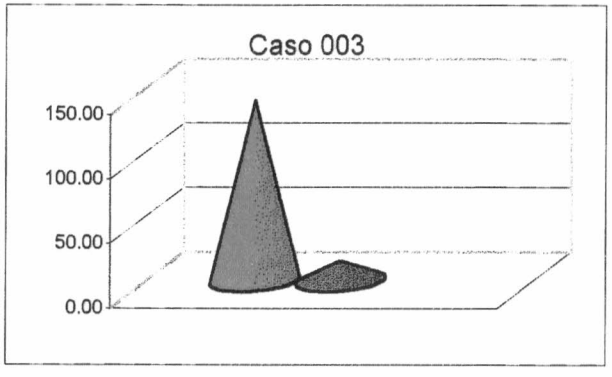
Referencia:

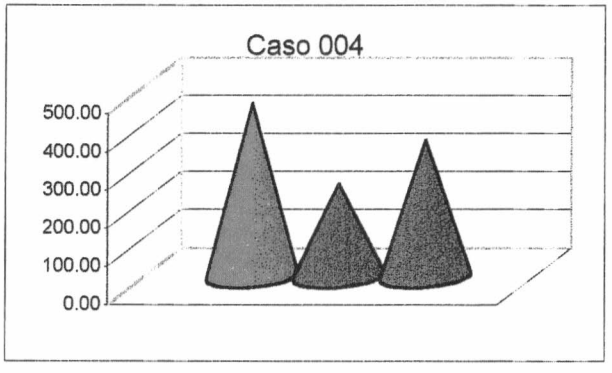
Algoritmos:

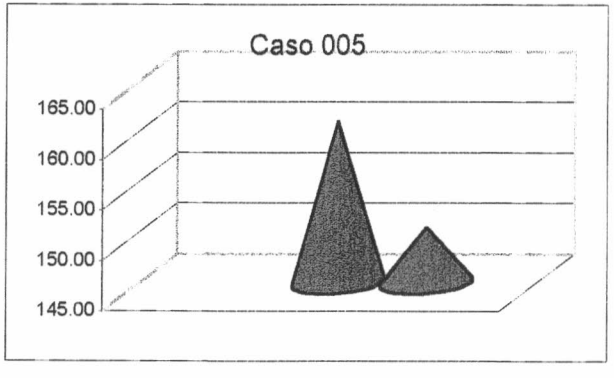
- Deadline Driven
- Deadline Driven con Aviso de Overload
- Deadline Driven con Aviso de Perdida de Meta

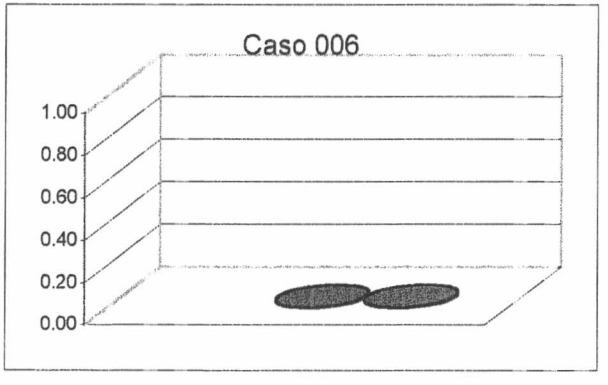

Caso 001
400.00
300.00
200.00
100.00
0.00

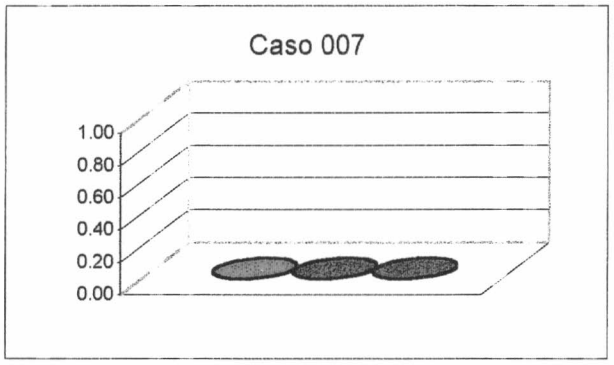

Caso 002
500.00
400.00
300.00
200.00
100.00
0.00

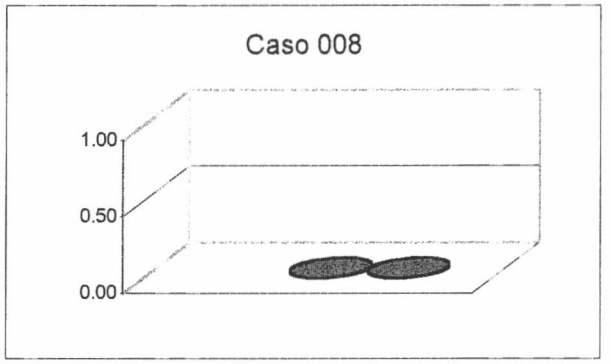

Caso 003
150.00
100.00
50.00
0.00

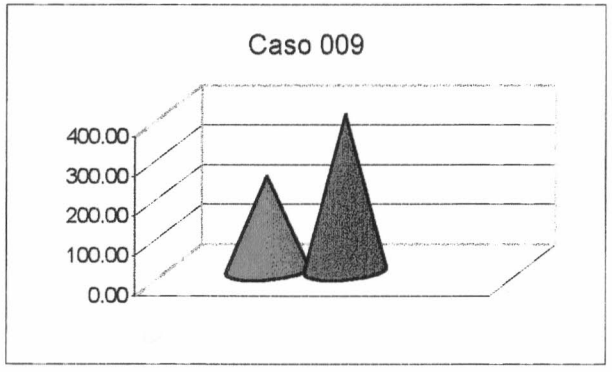

Caso 004
500.00
400.00
300.00
200.00
100.00
0.00

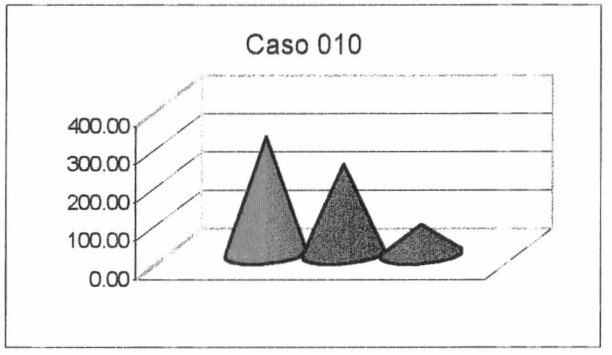

Caso 005
165.00
160.00
155.00
150.00
145.00

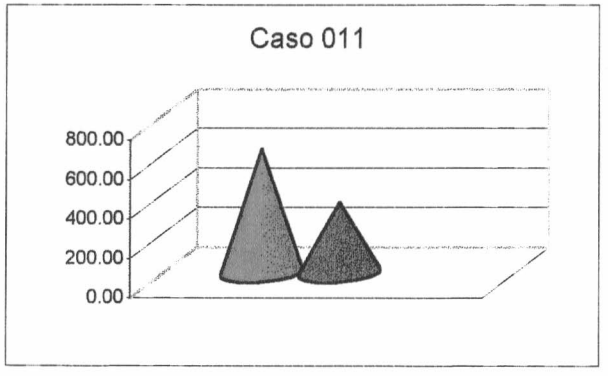

Caso 006
1.00
0.80
0.60
0.40
0.20
0.00

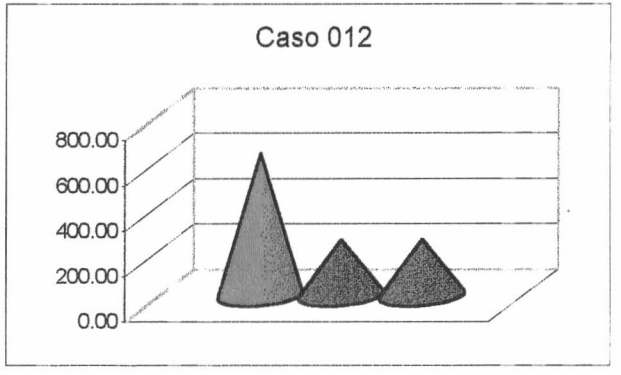

Caso 007
1.00
0.80
0.60
0.40
0.20
0.00

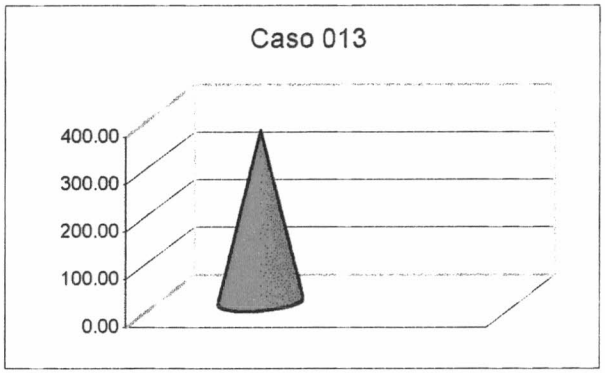

Caso 008
1.00
0.50
0.00

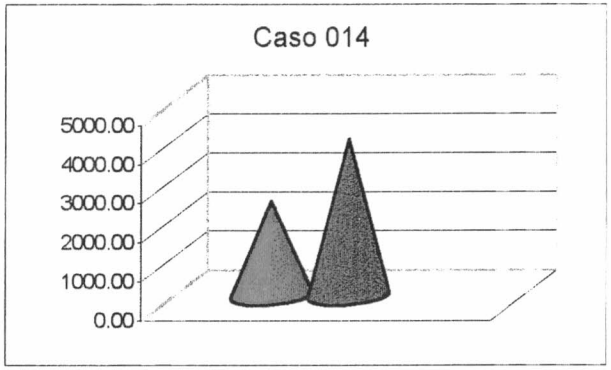

Caso 009
400.00
300.00
200.00
100.00
0.00

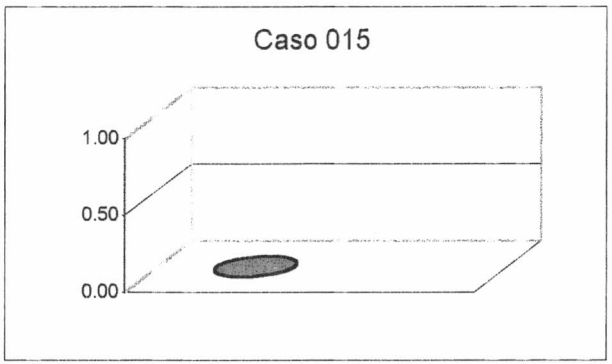

Caso 010
400.00
300.00
200.00
100.00
0.00

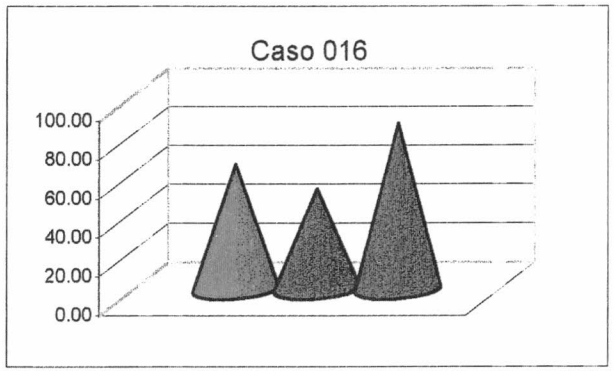

Caso 011
800.00
600.00
400.00
200.00
0.00

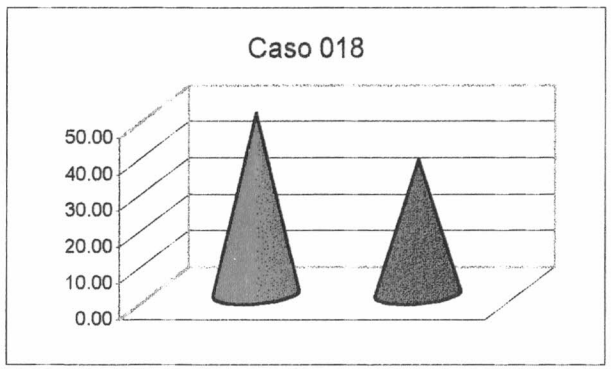

Caso 012
800.00
600.00
400.00
200.00
0.00

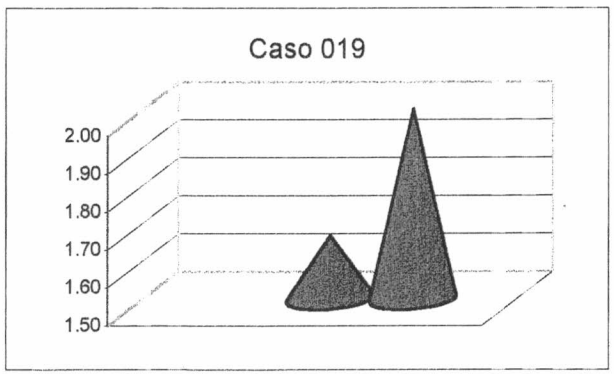

Caso 013
400.00
300.00
200.00
100.00
0.00

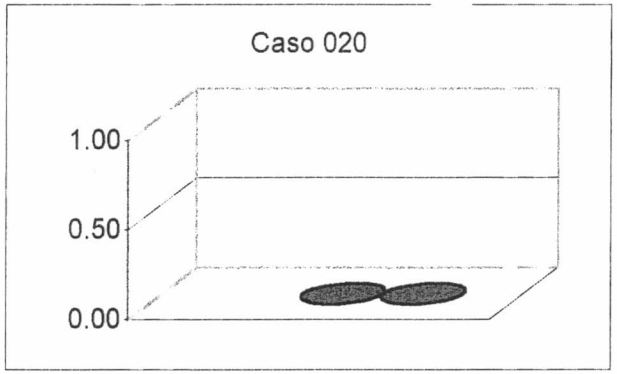

Caso 014
5000.00
4000.00
3000.00
2000.00
1000.00
0.00

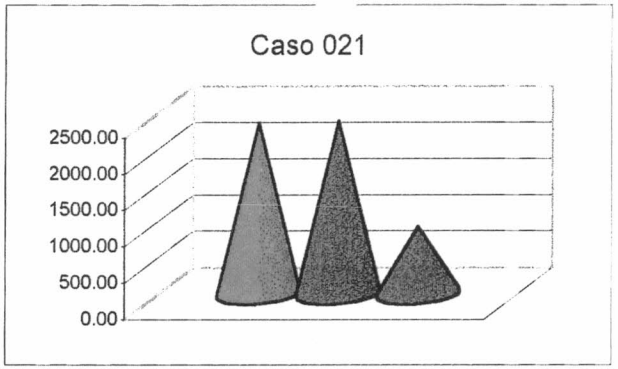

Caso 015
1.00
0.50
0.00

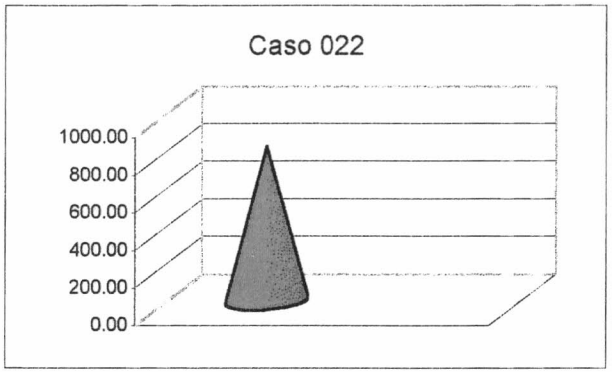

Caso 016
100.00
80.00
60.00
40.00
20.00
0.00

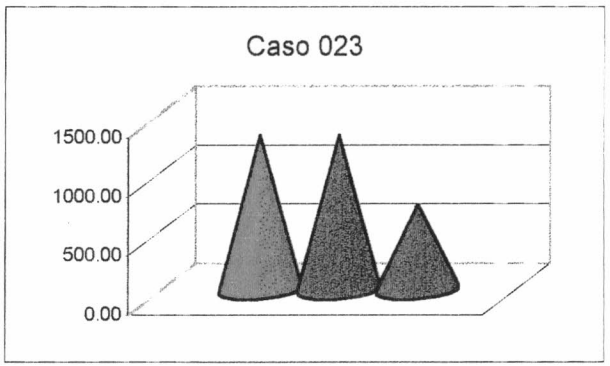

Caso 018
50.00
40.00
30.00
20.00
10.00
0.00

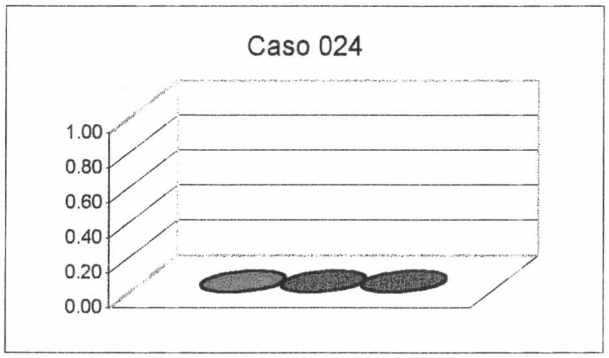

Caso 019
2.00
1.90
1.80
1.70
1.60
1.50

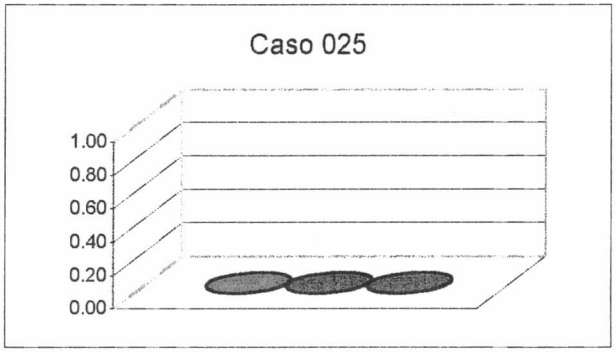

Caso 020
1.00
0.50
0.00

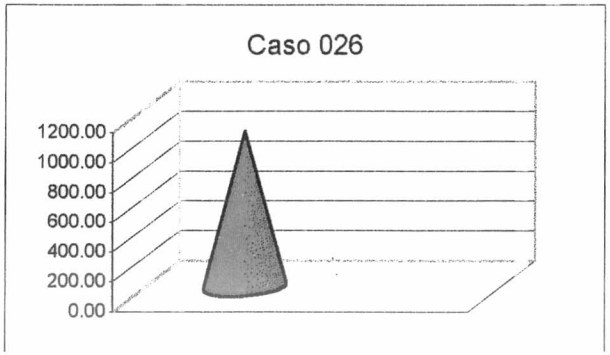

Caso 021
2500.00
2000.00
1500.00
1000.00
500.00
0.00


Caso 022
1000.00
800.00
600.00
400.00
200.00
0.00


Caso 023
1500.00
1000.00
500.00
0.00


Caso 024
1.00
0.80
0.60
0.40
0.20
0.00


Caso 025
1.00
0.80
0.60
0.40
0.20
0.00


Caso 026
1200.00
1000.00
800.00
600.00
400.00
200.00
0.00

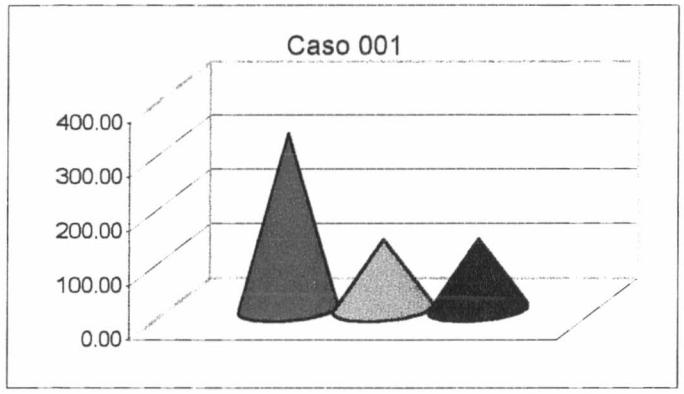
Polimorfismo de Desempeño – Tasa Monotónica

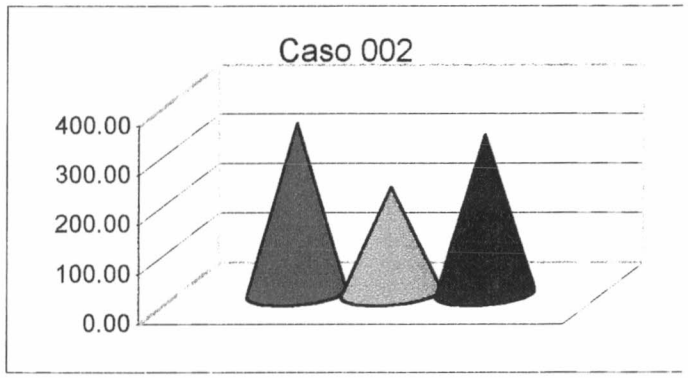
Los siguientes gráficos muestran la relación entre Tasa Monotónica y las dos funciones de polimorfismo que se testearon para cada caso.

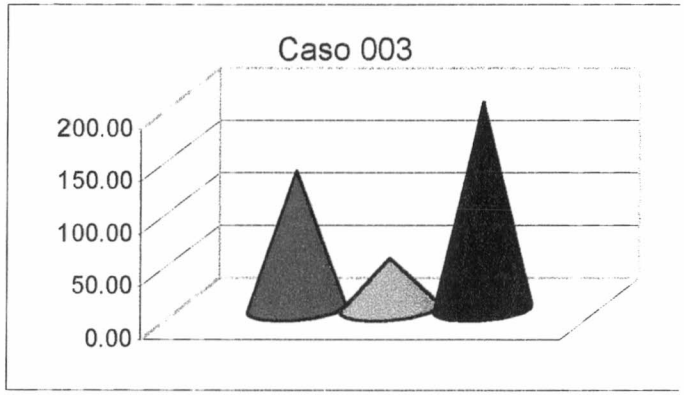
Referencia:

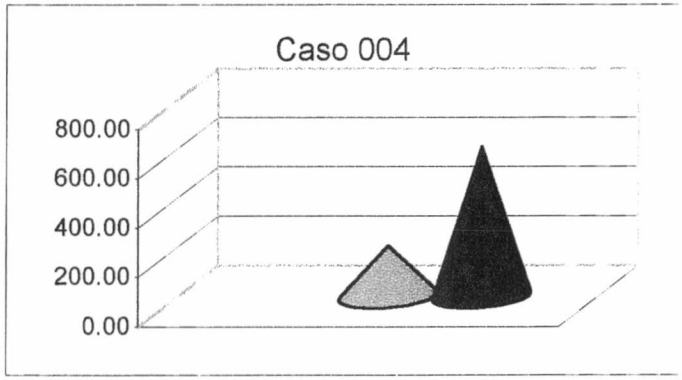
Algoritmos:

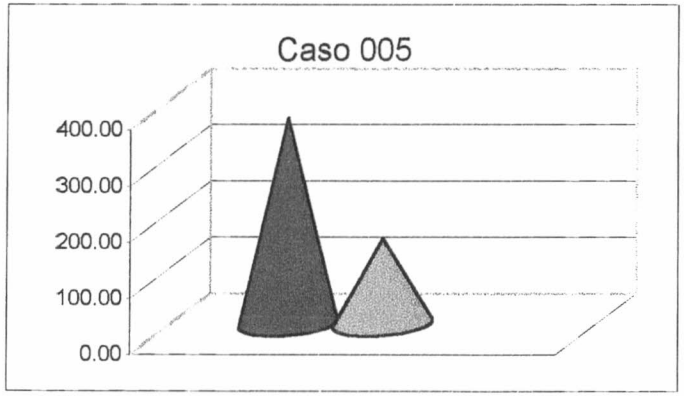
- Tasa Monotónica
- Tasa Monotónica con Aviso de Overload
- Tasa Monotónica con Aviso de Perdida de Meta

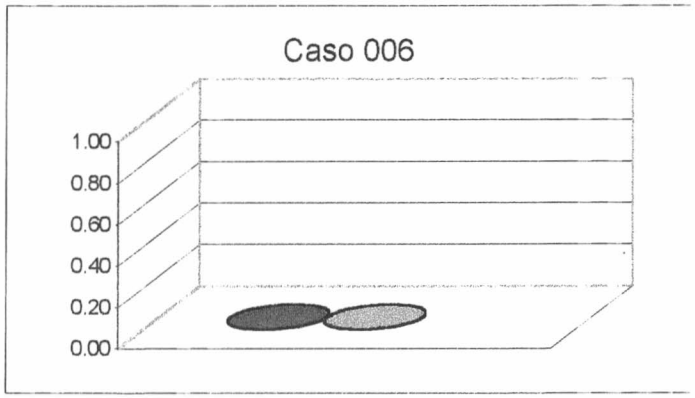

Caso 001
400.00
300.00
200.00
100.00
0.00

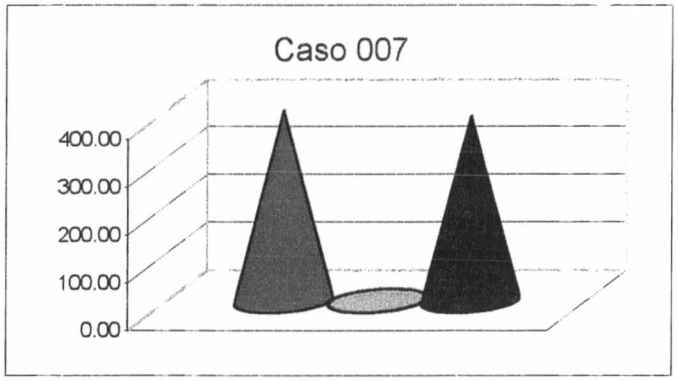

Caso 002
400.00
300.00
200.00
100.00
0.00

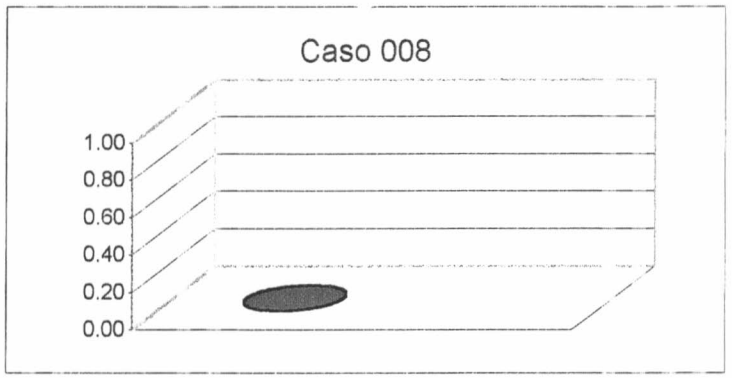

Caso 003
200.00
150.00
100.00
50.00
0.00

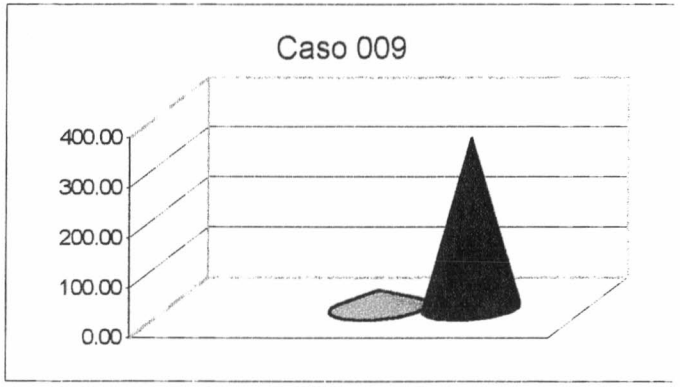

Caso 004
800.00
600.00
400.00
200.00
0.00

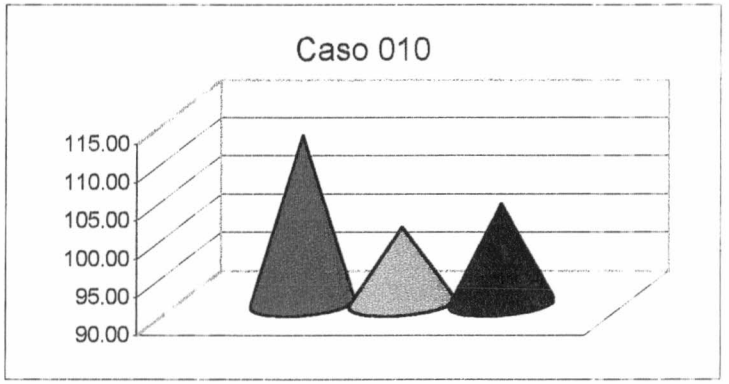

Caso 005
400.00
300.00
200.00
100.00
0.00

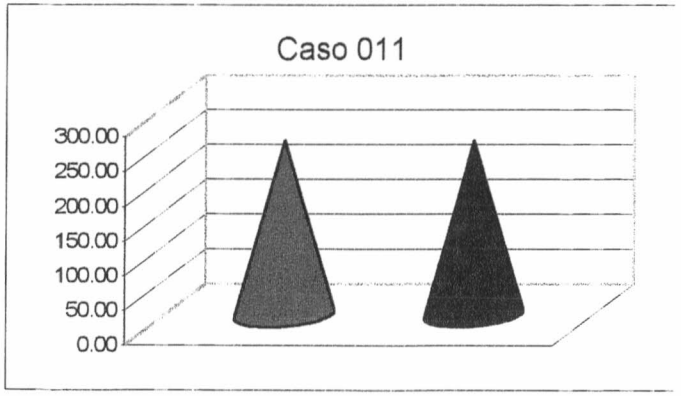

Caso 006
1.00
0.80
0.60
0.40
0.20
0.00

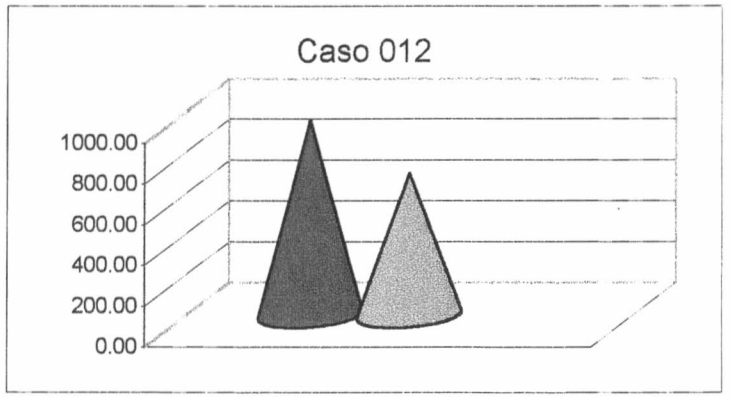

Caso 007
400.00
300.00
200.00
100.00
0.00

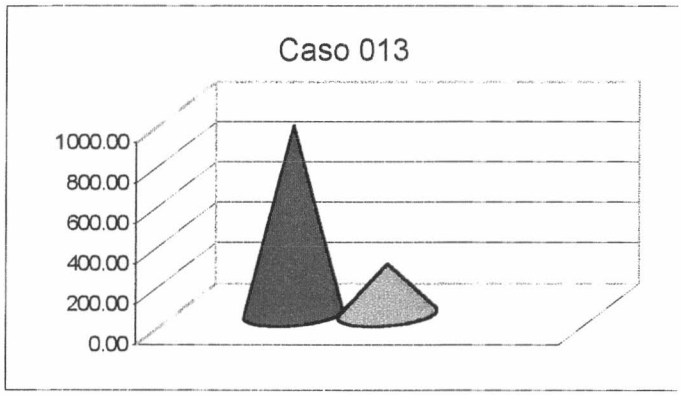

Caso 008
1.00
0.80
0.60
0.40
0.20
0.00

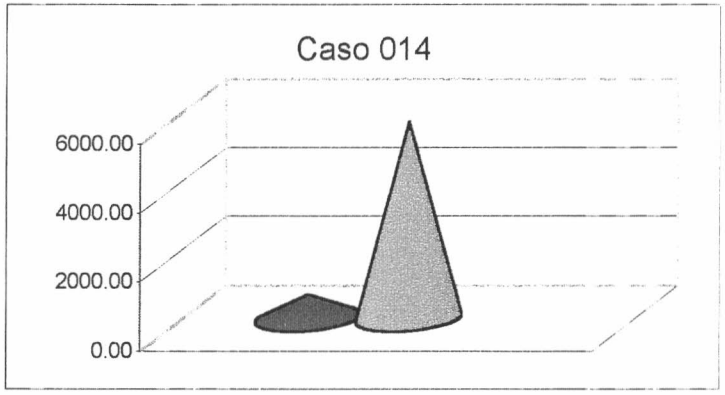

Caso 009
400.00
300.00
200.00
100.00
0.00

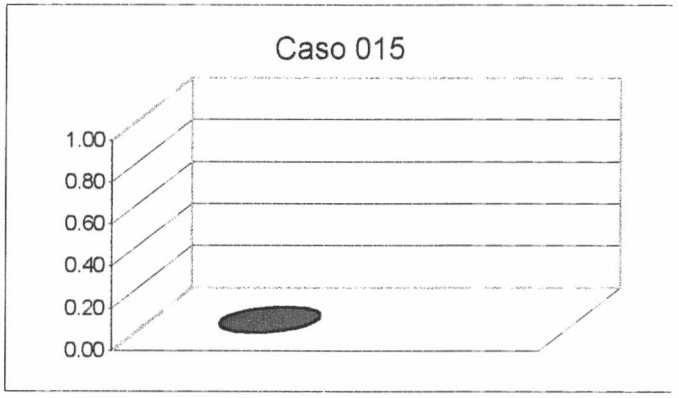

Caso 010
115.00
110.00
105.00
100.00
95.00
90.00

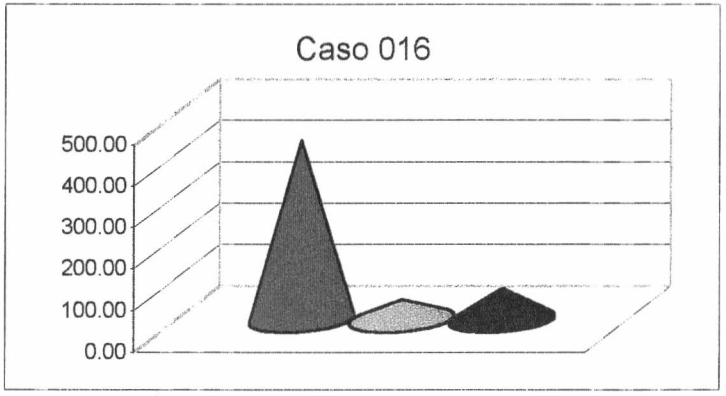

Caso 011
300.00
250.00
200.00
150.00
100.00
50.00
0.00

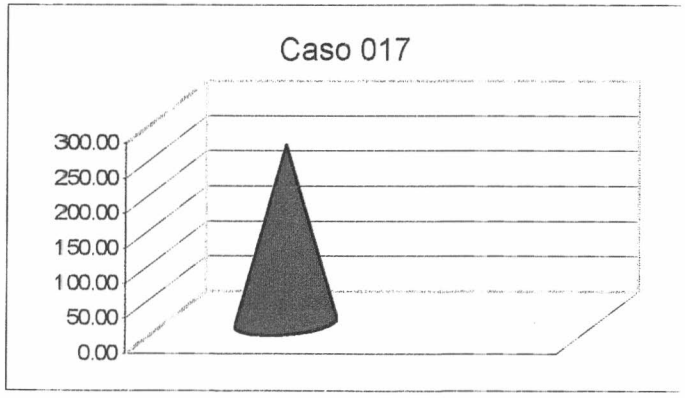

Caso 012
1000.00
800.00
600.00
400.00
200.00
0.00

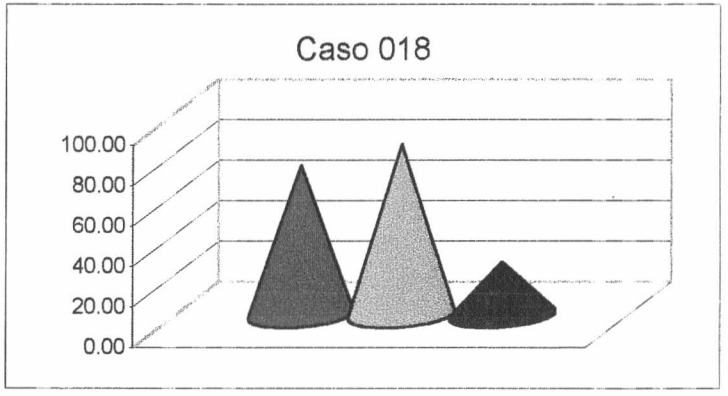

Caso 013
1000.00
800.00
600.00
400.00
200.00
0.00

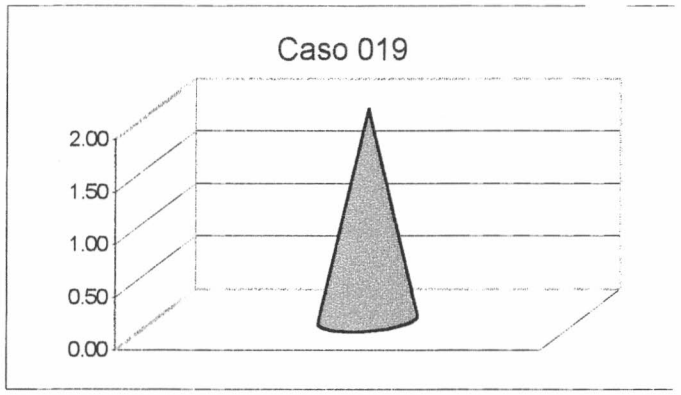

Caso 014
6000.00
4000.00
2000.00
0.00

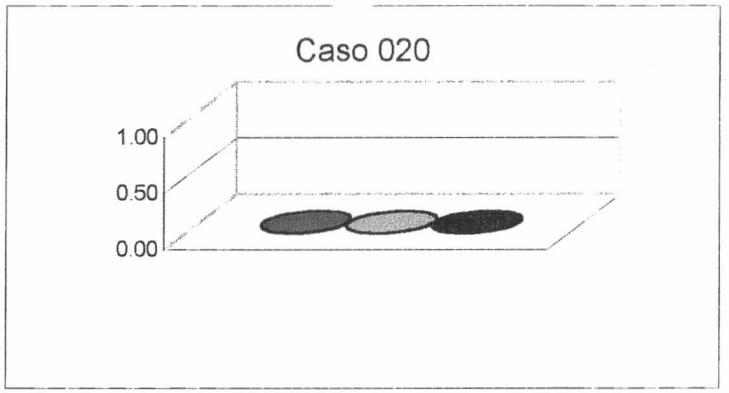

Caso 015
1.00
0.80
0.60
0.40
0.20
0.00

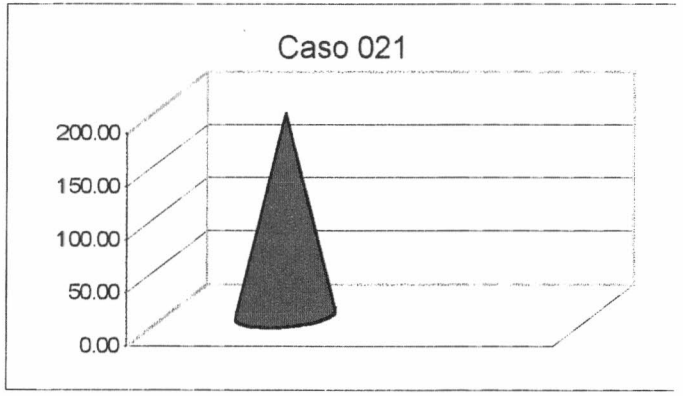

Caso 016
500.00
400.00
300.00
200.00
100.00
0.00

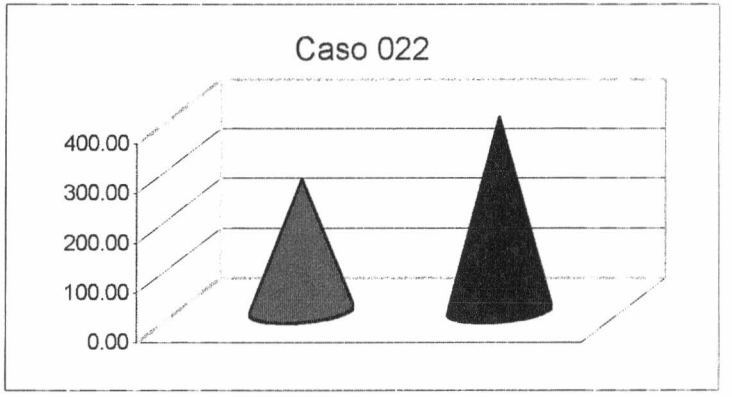

Caso 017
300.00
250.00
200.00
150.00
100.00
50.00
0.00

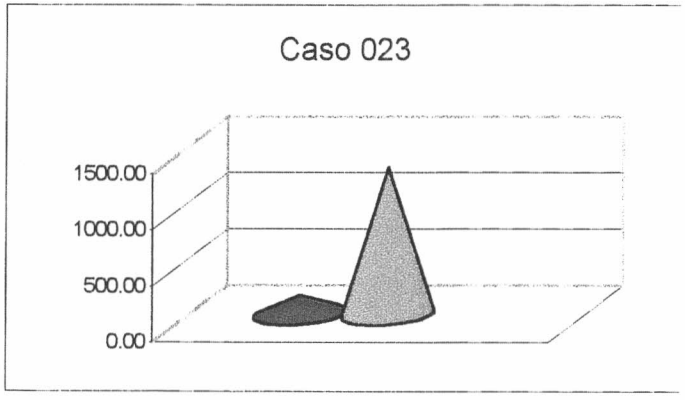

Caso 018
100.00
80.00
60.00
40.00
20.00
0.00

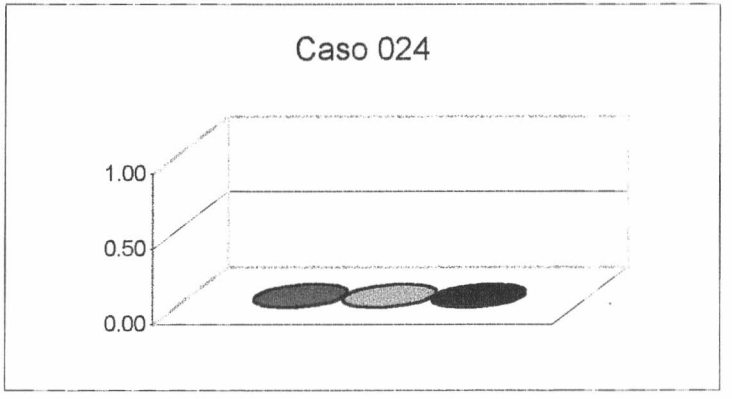

Caso 019
2.00
1.50
1.00
0.50
0.00

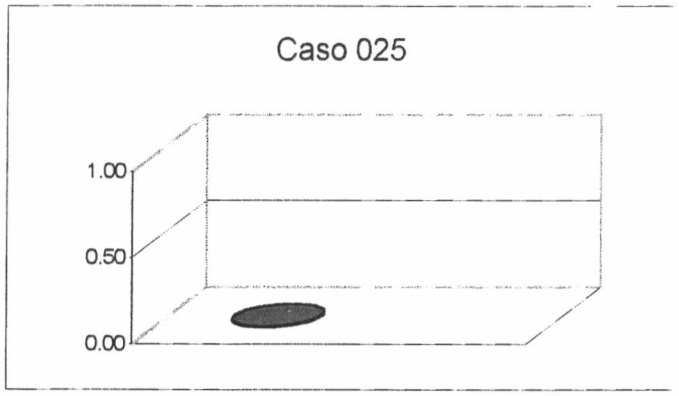

Caso 020
1.00
0.50
0.00

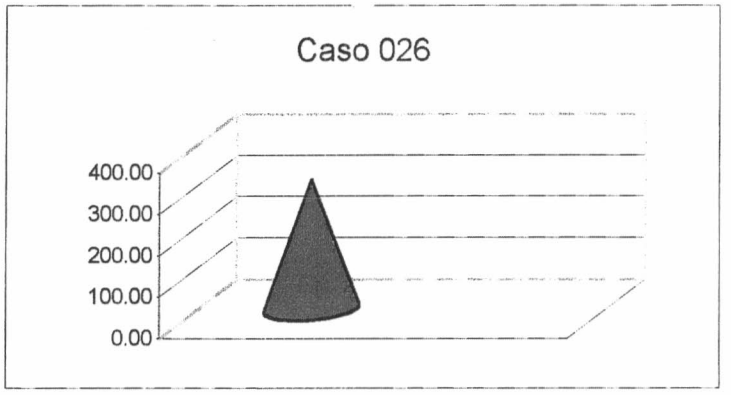

Caso 021
200.00
150.00
100.00
50.00
0.00


Caso 022
400.00
300.00
200.00
100.00
0.00


Caso 023
1500.00
1000.00
500.00
0.00


Caso 024
1.00
0.50
0.00


Caso 025
1.00
0.50
0.00


Caso 026
400.00
300.00
200.00
100.00
0.00